UNIVERSITA' DEGLI STUDI DI PAVIA

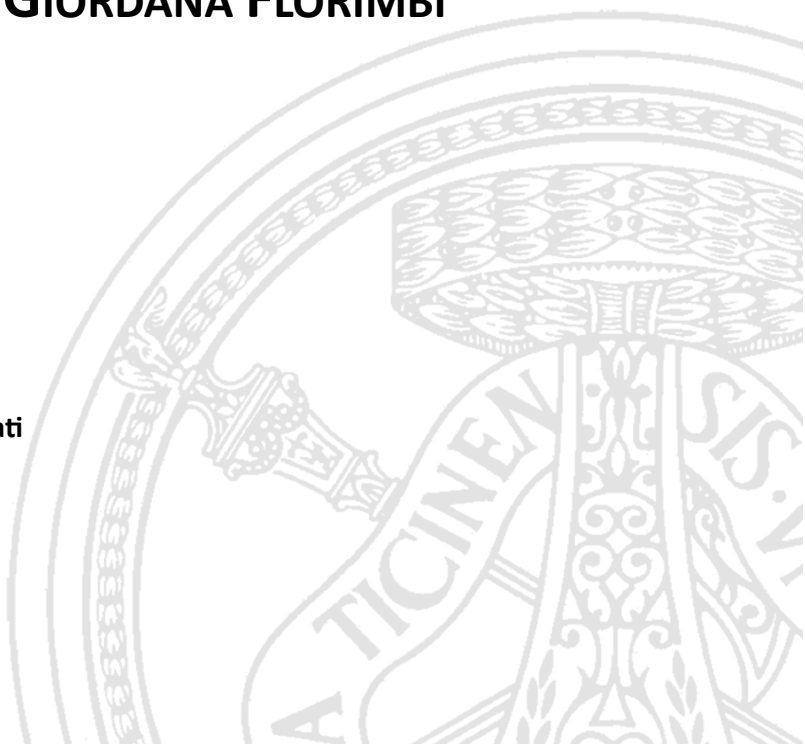
FACOLTA' DI INGEGNERIA
DIPARTIMENTO DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE

DOTTORATO DI RICERCA IN BIOINGEGNERIA E BIOINFORMATICA
XXXI CICLO - 2018

HIGH PERFORMANCE MODELLING AND COMPUTING IN COMPLEX MEDICAL CONDITIONS: REALISTIC CEREBELLUM SIMULATION AND REAL-TIME BRAIN CANCER DETECTION

PhD Thesis by

GIORDANA FLORIMBI



Advisor:
Prof. Francesco Leporati

PhD Program Chair:
Prof. Riccardo Bellazzi

Science is magic that works.

Kurt Vonnegut

Abstract (English)

The *personalized medicine* is the medicine of the future. This innovation is supported by the ongoing technological development that will be crucial in this field. Several areas in the healthcare research require performant technological systems, capable to elaborate huge amount of data in reduced computational times or even in real-time. By exploiting the *High Performance Computing* (HPC) technologies, researchers and scientists want to reach the goal of this innovative medicine: to develop specific and accurate diagnosis and personalized therapies for patients. To reach these goals three main activities have to be investigated: managing a great amount of data acquisition and analysis, designing computational models to simulate the patient clinical status, and developing medical support systems to provide fast decisions during diagnosis or therapies. These three aspects are strongly supported by several technological systems and devices that, at first glance, could appear different and disconnected. However, in this new innovative medicine, they will be in some way connected. As far as the data are concerned, today people are immersed in technology and, each of us is able to produce a huge amount of heterogeneous data. Part of these is characterized by a great medical potential that could facilitate the delineation of the patient health condition. Let's consider, as example, the information that we can produce using smartphones and smartwatches: these devices are even capable to monitor the heart electric activity that

could be shared with the medical team. Part of these heterogeneous data could be integrated in our medical record facilitating clinical decisions. To ensure this process, in addition to the data acquisition devices, technological systems able to organize, analyse and share these information are needed. Furthermore, they should guarantee a fast or real-time data usability. In this contest the HPC systems will surely have a high importance and, in particular, the ones characterized by multicore and manycore processors, capable to spread the computational workload on different cores of the device in order to reduce the elaboration times. These technological solutions are crucial also in the computational modelling, second aspect cited above and very important in the personalized medicine. Several research groups and a lot of projects funded in the medical area, aim to implement models able to realistically reproduce the human organs behavior in order to develop their simulators. They are called *digital twins* and allow to reproduce the organ activity of a specific patient in order to study the disease progression or a new therapy. Patient data will be the inputs of these models which will predict her/his condition, avoiding invasive and expensive exams. The technological support that a realistic organ simulator requires is significant from the computational point of view. HPC supports this research field where complex equations systems have to be solved as fast as possible. For this reason, devices as *Graphics Processing Units* (GPUs), *Field Programmable Gate Arrays* (FPGAs), multicore devices or even supercomputers are needed to sustain such a computational load and to guarantee fast or real-time simulations. As an example in this research field, the development of a cerebellar simulator exploiting HPC technologies will be described in the second chapter of this thesis. It will be possible to understand the complexity of the realistic mathematical models that will justify such a technological choice to reach reduced elaboration times, aiming at real-time. This work is within the european *Human Brain Project* that aims to run a complete simulation of the human brain, exploiting models which can realistically reproduce its physiological behavior.
Finally, these technologies have a crucial role in the medical support system development. Most of the times in medicine, especially during surgeries, it is very important that a support system provides a real-time answer. More-

over, the fact that this answer is the result of the elaboration of a complex mathematical problem, makes HPC system essential also in this field. If environments such as surgeries are considered, it is more plausible that the computation is performed by local *desktop* systems, such as multi-GPU systems, able to elaborate the data directly acquired during the surgery.
The third chapter of this thesis describes the development of a brain cancer detection system, exploiting the GPU technology. This support system, developed as part of the *HypErspectraL Imaging Cancer Detection* (HELICoiD) project, performs a real-time elaboration of the brain hyperspectral images, acquired during surgery, to provide a classification map which highlights the tumor tissue in the image. In this way the neurosurgeon is facilitated in the tissue resection. In this field, the GPU technology has been crucial to provide a real-time elaboration that a processor was not able to provide.
Finally, it is possible to assert that in most of the fields of the personalized medicine, HPC technologies will have a crucial role since they consist in the elaboration of a great amount of data in reduced times, aiming to provide specific diagnosis and therapies for the patient. In this context, this thesis describes examples where these technologies are crucial for the develoment of cerebellar computational models and of support system for the brain cancer detection.
In Chapter 1 the use of HPC technologies in medicine is described. Furthermore, the parallel technologies exploited during the work will be presented.
Chapter 2 presents the HPC technologies relevance in the development of algorithms, based on realistic models, to reproduce the cerebellar granular layer neuronal activity. This work is within the Human Brain Project and it has been realized in the Custom Computing Programmable System laboratory (University of Pavia), in collaboration with IRCCS Istituto Neurologico Nazionale C. Mondino (Pavia) and the Brain Connectivity Center (Pavia).
Chapter 3 presents a HPC solution to the brain cancer detection system presented above. This work has been performed at the University of Las Palmas de Gran Canaria (ULPGC) and it is part of the HELICoiD project.
Finally, chapter 4 describes the overall conclusions of the work.

Abstract (Italian)

La *medicina personalizzata* è la medicina del futuro. E questo cambiamento è supportato dal continuo sviluppo tecnologico che gioca un ruolo fondamentale. Diversi ambiti della ricerca medica richiedono ormai in modo imprescindibile sistemi tecnologici ad alte prestazioni, capaci di elaborare grandi quantità di dati in tempi ridotti o persino in real-time. Ed è proprio sfruttando questa tecnologia chiamata *High Performance Computing* (HPC) che ricercatori e scenziati possono raggiungere l'obiettivo della medicina moderna: sviluppare diagnosi specifiche ed accurate per pazienti e delineare terapie altamente personalizzate. Per raggiungere questi scopi, una grande quantità di dati deve essere acquisita ed analizzata, adeguati modelli computazionali devono essere sviluppati ed utilizzati per simulare la condizione clinica del paziente e sistemi di supporto per medici devono essere implementati per fornire risposte rapide in fase di diagnosi e terapia. Questi tre aspetti sono fortemente legati a diversi dispositivi e sistemi tecnologici che possono sembrare lontani ma, che nella medicina del futuro, saranno in qualche modo connessi. Per quanto riguarda i dati, ad oggi siamo immersi nella tecnologia e, quindi, ognuno di noi è in grado di produrre una gran quantità di dati eterogenei. Di questi, molti hanno un grosso potenziale in ambito medico che potrebbe agevolare la definizione della condizione di salute del paziente. Si pensi, ad esempio, a tutta l'informazione che siamo capaci di produrre sfruttando i nostri *smartphone* o

smartwach: questi dispositivi sono in grado, ormai, di monitorare persino l'attività elettrica del cuore che potrebbe essere condivisa con il nostro medico curante. Una selezione di questi dati eterogenei può arricchire la nostra cartella clinica agevolando possibili decisioni mediche. Per garantire tutto ciò, oltre alla tecnologia che acquisisce questi dati, vi è necessità di sistemi tecnologici in grado di organizzare, analizzare e condividere questa grande quantità di informazione e che possano garantire una fruibilità real-time. Sicuramente in questo contesto una grande rilevanza l'avranno sistemi HPC ed, in particolare, quelli caratterizzati da processori multicore e manycore che permettono di suddividere il carico computazionale su diversi core del dispositivo per ridurre i tempi di elaborazione. Sono proprio queste soluzioni tecnologiche che sono fondamentali nello sviluppo di modelli computazionali, secondo aspetto citato prima di elevatissima importanza nella medicina personalizzata. L'obiettivo di molti gruppi di ricerca e di molti dei grandi progetti finanziati in abito medico, è quello di sviluppare modelli che riproducano in modo realistico il comportamento di organi in modo da crearne veri e propri simulatori. Questi vengono definiti *digital twins*, ossia gemelli digitali di organi che permettano di simulare la condizione di un organo di uno specifico paziente per studiarne il decorso di una malattia o una nuova specifica terapia. I dati relativi al paziente saranno forniti come input a questi modelli che prediranno la condizione del paziente evitando test invasivi e dispendiosi. Il supporto tecnologico che richiede un simulatore realistico di un organo è molto grande in termini di potenza computazionale. Le tecnologie HPC supportano questo ambito della ricerca dove modelli caratterizzati da complessi sistemi di equazioni devono essere risolti il più velocemente possibile. Ed è per questo che i modelli computazionali richiedono quelle tecnologie come *Graphics Processing Units* (GPUs), *Field Programmable Gate Arrays* (FPGAs), dispositivi multicore o persino supercomputer che possano sostenere un tale peso computazionale e garantire simulazioni real-time. In questo contesto, verrà presentato lo sviluppo di simulatore di cervelletto su tecnologie ad alte prestazioni nel secondo capitolo di questa tesi. Sarà possibile apprezzare la complessità dei modelli matematici realistici che giustificano tale scelta tecnologica per ottenere simulazioni a ridotto tempo di elaborazione, mirando al real-time.

Tale lavoro è inserito nel progetto europeo *Human Brain Project* che mira a compiere una simulazione completa del cervello umano sfruttando modelli che ne sappiano riprodurre i comportamenti fisiologici in modo realistico.
Infine, le tecnologie ad alte prestazioni hanno un ruolo fondamentale anche nello sviluppo di sistemi di supporto per i medici. Molte volte in medicina, specialmente in sede di intervento operatorio, è di cruciale importanza che un sistema di supporto fornisca una risposta in tempo reale. Ma il fatto che questa risposta sia il frutto di elaborazione di un complesso problema computazionale fa sì che sistemi HPC vengano sfruttati. Se si ha a che fare con contesti come inteventi operatori, è più plausibile che il calcolo debba essere sostenuto da sistemi *desktop* locali, come ad esempio sistemi multi-GPU, che possano elaborare il dato acquisito direttamente in sala operatoria. Un altro capitolo di questa tesi riguarderà proprio lo sviluppo su tecnologia GPU di un sistema di individuazione del cancro al cervello. Questo sistema di supporto, sviluppato nell'ambito del progetto *HypErspectraL Imaging Cancer Detection* (HELICoiD), elabora in tempo reale immagini iperspettrali del cervello, acquisite durante un intervento operatorio, per fornire una mappa di classificazione che individui il tessuto cancerogeno nell'immagine in modo che il medico possa asportarlo con accuratezza. In questo contesto, la tecnologia GPU è stata fondamentale per una risposta in tempo reale che un processore non era in grado di fornire.
Quindi è possibile dire che in ogni ambito della medicina personalizzata le tecnologie HPC avranno un ruolo fondamentale perchè essa si baserà sull'elaborazione di una grande quantità di dati in tempi ridotti, con l'obiettivo di fornire diagnosi e terapie specifiche per il paziente. In questo contesto, il lavoro presentato fornisce un esempio di come queste tecnologie siano essenziali nello sviluppo di modelli computazionali cerebellari e di un sistema di detenzione di cancro cerebrale. Nel Capitolo 1 viene introdotto l'uso di tecnologie HPC in medicina e vengono presentate le tecnologie parallele utilizzate durante i progetti descritti.
Il Capitolo 2 descrive l'importanza dell'uso di tecnologie HPC nello sviluppo di algoritmi, basati su modelli realistici, per la riproduzione dell'attività neuronale dello strato granulare del cervelletto. Questo lavoro è stato sviluppato all'interno del progetto europeo Human Brain Project presso il

laboratorio Custom Computing Programmable System (Università di Pavia), in collaborazione con IRCCS Istituto Neurologico Nazionale C. Mondino (Pavia) e il Brain Connectivity Center (Pavia).
Il Capitolo 3 presenta, invece, una soluzione HPC per il sistema di identificazione di tumore cerebrale citato prima. Questo lavoro è stato sviluppato presso l'Universidad de Las Palmas de Gran Canaria (ULPGC) ed è parte del progetto HELICoiD.
Infine, il Capitolo 4 presenterà le conclusioni generali del lavoro.

Contents

List of Figures

List of Tables

Chapter 1

High Performance Computing in medicine

Nowadays people's life is completely filled with technology. Everyday a person can produce, share and collect a huge amount of heterogeneous data thanks to technological devices that people can even wear. One of the main challenges of these days is to integrate these data to improve people's quality of life. This challenge has been accepted by the healthcare and the medical community with the intempt to exploit these data to define an innovative way of medicine, called *personalized* or *precision medicine*. The idea is to collect genetic information, electronic health records (EHRs), medical imaging and computational modelling to generate very accurate diagnosis, to determine the best therapy for a specific patient and to prevent her/him from possible future diseases. Furthermore, considering that in most of these cases the data acquisition and elaboration have to be real-time compliant, it is clear that to carry out a plan so ambitious, a complex and efficient technological support is needed. High Perfomance Computing (HPC) plays a crucial role in this innovative field since it provides the means to efficiently collect, analyse and share data in order to improve the quality of the medical decisions and to facilitate the researchers' work.

1.1 Introduction

The High Performance Computing (HPC) is transforming the traditional medicine into *precision medicine* which is moving from a *one-size-fits-all* clincal approach to a faster, more efficient model of patient-centric research [1]. In fact, this innovative way of medicine aims to generate very accurate diagnosis and to select personalized therapies for a specific patient. The innovation which is the basis of this transformation is the growing data collection which is facilitated by the ongoing technological development. In the medical area, the challenge is to find a way to collect, organize and share data in order to exploit them in a constructive way as well as provide a new approach to medical research. In this innovative context, the computational modelling plays a crucial role since it is entering in every aspects of medicine and biology. Indeed, several research groups are studying organs in order to develop models that realistically reproduce their behaviours and functionalities so they can help researchers to better understand how they work. In fact, simulation allows to explore a procedure or an innovative technique in a virtual environment and analyse its consequences instead of studying them in vivo, for example trying some invasive and expensive exams or therapy directly on the patient [1]. It also helps clinicians and researchers in the reconstruction of the patients' anatomy providing more knowledge on his disease progression and the prediction of the biological response as well. In addition, simulating an organ or a biological system can lead to the choice of the best drug, device or surgery for a specific patient [2]. For these reasons, the attention of the scientific community is so high on these aspects that some of the most important projects aim at building realistic simulators of human organs. The *Living Heart Project* [2], for example, created a 3D realistic simulation of a human heart. The virtual model behaves like a real organ allowing scientists to study new pathologies and test new therapies, trying to reduce incorrect diagnosis or outcomes. Furthermore, a simulator allows patients to better understand the surgery or the therapy that they will face and it provides surgeons with a tool for a more realistic study and preparation of the operation. These aims have been reached thanks to the developed virtual reality environment

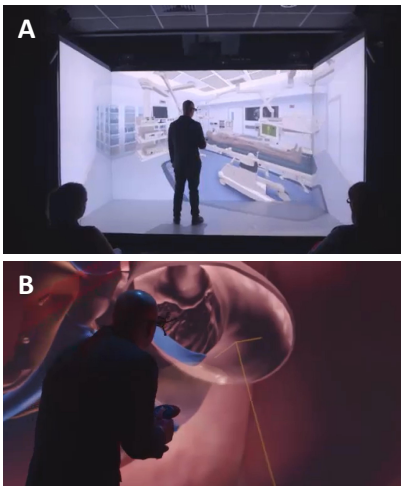


Figure 1.1: The Living Heart Project. A scientist is exploring the surgery scenario (A) and and the heart (B) thanks to the virtual reality [2].

which reproduces the human body (Figure 1.1.A) in a surgery scene and the heart (Figure 1.1.B). Another important project which follows this research philosopy is the *Human Brain Project* whose aim is to provide a complete simulation of the human brain [3]. Several european research groups are studying different parts of the brain, at different levels of organization, in order to reach a complete knowledge of this organ. Their aim is to develop models that reproduce neurons functionalities in order to study new drugs, therapies, diseases and avoid invasive tests and experiments. These projects are working for the development of the so-called *digital twin* [4]: in the new personalized medicine the duplicate of a patients' organ will have a crucial role allowing doctors to explore the organ behaviour and increase the personalization of treatments, making better data-driven care decisions and preventing medical complications before they may occur [1]. The simulation is a powerful tool to integrate the knowledge and the experience of researchers and professionists in order to understand the single patient condition. Sometimes doctors have an indication of what the problem is and the research of the specific problem can be very difficult, expensive and

invasive. Another main aspect that characterized these projects is the data sharing: researchers want to make available models, tests, experiments in order to rapidly increase the knowledge of these complex organs so as potentially improve and save people's lives.
The genetic information could be also added to the computational modelling since they could provide useful data about the patient genomic profile: for example, if a patient is predisposed to a disease and lives in an area of the country where other people showed this pathology, maybe it could be useful to plan a primary prevention before any evidence of developing appears.
Another interesting information that could be integrated consists of all the data that are generated by medical devices, medical support systems and personal devices. All these data will be included in the routine medical care and will allow an easier data sharing between patient and medical team, providing a more accurate and always updated patient knowledge. Consider, for example, the amount of data that a smartphone or a smartwatch can provide nowadays: the ability of understanding if the user is sleeping, walking, running and to evaluate the heart rate during these actions. If it is considered that the most recent smartwatches can acquire even an ECG signal and send it to the user's medical team, it is clear that everyday a person can produce heterogeneous and useful data that, if well managed, could provide an always updated information to add to the medical records. This new way of making medicine leads researchers to deal with a range of issues such as the need of defining a standard protocol to organize and store data in order to facilitate the information sharing. Furthermore, it is crucial to evaluate ethical aspects to protect the patients' privacy.
The realisation of this huge innovative system is possible only if two leading factors are considered: a fast data availability and a technological infrastructure. Several tasks shown in Figure 1.2 required a fast data elaboration and in most cases a real-time execution. The computational modelling presented above requires technologies capable of elaborating complex equations systems which reproduce the cells or biological structures functionalities. Furthermore, if it is considered that those systems has to be solved several times on the base of the cells number simulated, it is clear that the computational load is very heavy. For this reason, these simulations require

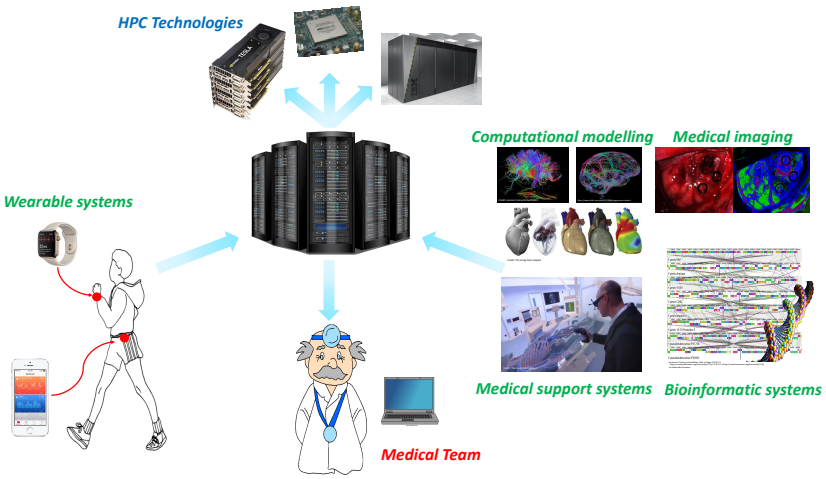


Figure 1.2: Data collection and sharing in the personalized medicine. The research groups which develop computational models, medical imaging devices, medical support systems and bioinformatic systems require the HPC technologies to reach fast elaborations or solutions managing complex models or systems. The medical team can access the results of the researchers' activity, together with the health data acquired from the wearable system.

technologies capable to perform complex computations as fast as possible, trying to satisfy the real-time constraints.
The Graphics Processing Units (GPUs) are widely used in the computational modelling field, since their architecture, characterized by thousands of cores, allows to spread the workload among these processors which work in parallel. Sometimes, researchers exploits multi-GPU systems or even supercomputers in order to increase the number of available cores, trying to further reduce the elaboration time. An example of neuronal modelling which exploits single-GPU and multi-GPU systems, part of the Human Brain Project, will be presented in the second chapter of this thesis, where it will be possible to appreciate how this technology can significantly reduce the computational time of the simulations.
These medical systems which support the doctors during a surgery providing them help in the diagnosis or in the medical decision need to satisfy the

real-time constraint. For example, the HELICoiD project [5] has developed a system which provides a support to neurosurgeons for the brain cancer detection. This system is capable to classify in real-time a brain hyperspectral image providing a classification map of that part of the organ where the tumor is accurately delineated. Also in this case the GPU technology is exploited for real-time elaboration.
In addition, applications concerning the medical imaging, as the magnetic resonance imaging (MRI) [6] [7] and the computed tomography (CT) [8] [9] image reconstruction algorithms, need to elaborate images in short times or even in real-time and for this reason, also in these cases, the GPU technology is used. Nowadays several research groups, for example the ones working in the fields cited above (also shown in Figure 1.2), exploit HPC technology using PC-desktop solutions, as single or multi-GPUs systems, or supercomputers, characterized by a huge number of processors or made up of GPUs or of programmable logic devices. The innovative personalized medicine aims to collect heterogeneous information as well as the health records acquired from wearable devices. In this way the medical team could benefit from several types of data to make accurate diagnosis and therapies and to reach all the goals described before. In this new perspective of work, a suitable technological infrastructures able to collect all these data should be developed. Some research group has already started following this information sharing phylosophy. For example, in the Human Brain Project, an infrastructure made up of six platforms [10] is developed with the aim of sharing research results. If this phylosophy will be followed by an increasing number of scientists, there should be also a suitable use of techology systems that will allow researchers to upload their results in servers and share data. Furthermore, the medical team or scientists could download and use them for diagnosis, therapies or research.
The scientific community is very focused on this issue that companies as Medtronic or IBM work on this aspect trying to find the best technological solution. Medtronic [11] [12], for example, declares that the current technologic system used to share data is disconnected and do not facilitate users to obtain data that they need. This problem leads to another issue which is the unusable information: in [13] authors assert that healthcare is los-

ing $300B per year in untapped data integration. Companies as Medtronic think that *blockchain* might be the solution. The idea is to spread the workload providing a decentralized system where a huge amount of technological blocks can share, verify, analyse data and freely transfer them in complete safety. IBM [14] is also exploiting blockchain as a solution to faster access data, facilitating better collaboration and overcoming problems as the data security, thanks to the cryptography, and standardization. Whatever will be the best solution to collect and share data in the most efficient way, HPC technology will surely have a crucial role in the personalized medicine of the future. And this future is not so far away.

1.2 Parallel Technologies

The number of medical applications that rely on the power of more than a single processor is increasing. Often in healthcare providing a fast or even real-time response is crucial. This aspect leads to the evaluation of efficient technologies able to manage huge amount of data in reduced computational times. The use of the HPC and, in particular, of the parallel technologies allows to overcome this issue making several processors available where the workload can be splitted. As a consequence, a different way of programming is exploited in order to transform a *sequential* in *parallel* application, where different threads collaborate to rapidly end the elaboration. Two philosophies are adopted to exploit the parallel technologies. In the *multicore* strategy, the processor elaborates both the serial and the parallel parts of the code. The former are executed by a thread while the latter by several threads created by the operative system and assigned to the different cores. In this type of solutions each core has to run several threads, in the *manycore* philosophy the idea is that each core has to manage one thread, since the these technologies host hundreds or thousands of cores.

1.2.1 OpenMP

Concerning the multicore strategy, the Application Programming Interface (API) OpenMP is a parallel programming model for shared memory

multiprocessors which provides a wide set of directives and strategies for the parallelization of loops and program sections [15] [16]. It supports Fortran, C and C++ languages.
OpenMP implements the *multithreading* strategy (Figure 1.3) where a *master thread* executes the sequential code until a *parallel region* is generated. In this point, several threads (*slave threads*) are created on the base of the tasks that have to be completed. To use OpenMP in a C or C++ code the *omp.h* library has to be included. The definition of a parallel region, shown in Figure 1.3, is implemented through the *#pragma* directive. If the loop iterations have to be parallelized, this directive is placed before the loop as shown in the following code:

```
1 #pragma omp parallel for
2 for (int i = 0; i < n_iter; i++) {
3       ...
4 }
```

In this way a group of iterations is executed by a thread generated in correspondence of the parallel region creation. The number of slave threads can be set by the user exploiting the *omp_set_num_threads* function. Furthermore, when the parallel region is created, is important to define which

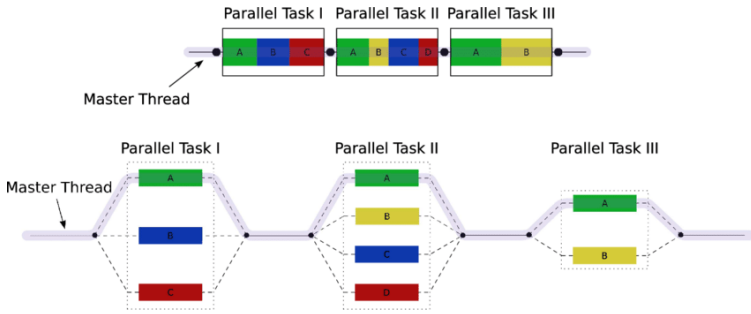


Figure 1.3: Multithreading OpenMP [17].

variables are *shared* among threads or which are *private* to a single thread. The manner in which iterations of a parallel loop are assigned to a thread is called *schedule*. OpenMP supports different schedulers [18]:

- *static*: the iterations are divided into *chunk* of size *chunk-size*. The operative system divides the loop iterations as equal as possible in the case the iterations number is not evenly divisible by the threads number. The chunks are distributed among threads in a circular order.
- *dynamic*: each thread executes a chunk of iterations and then requests other chunks until they are available. There is not a particular order in the chunk distribution. Furthermore, the chunk-size assumes the value one, if it is not explicitly specified.
- *guided*: it works as the dynamic scheduling but the chunk-size is different since it is proportional to the number of unassigned iterations divided by the number of threads. In this way the chunk-size decreases. If the chunk-size is not set, its value is one.
- *auto*: the scheduling to apply is chosen by the compiler.
- *runtime*: the *OMP_schedule* environment variable specifies which one of the three loop-scheduling types should be used.

OpenMP is also used to distribute different tasks to different threads. The *section* construct is a way to implement this aspect. The following code contains the instructions to generate three sections, each one assigned to a thread which executes the task.

```
1 #pragma omp parallel
2 {
3     #pragma omp sections
4     {
5         #pragma omp section
6         {
7             // task 1
8         }
```

```
 9           #pragma omp section
10           {
11               // task 2
12           }
13           #pragma omp section
14           {
15               // task 3
16           }
17           ...
18       }
19   }
```

All the sections are independent and their tasks are computed simultaneously.

1.2.2 Graphics Processing Units

CPU vs GPU architecures

Concerning the manycore philosophy, the Graphics Processing Units (GPUs) are the flagship devices for parallel high performance computing. They are born as graphic accelerators but researchers and scientists rapidly began to use their intrinsic parallel architecture for general purpose computing [19]. Figure 1.4 shows the evolution of the peak double precision floating-point operations per second (FLOPS) on GPU architectures versus the CPU ones. This graph is not a real performance comparison since it does not necesserarily represent the application speed, but it is possible to appreciate how much the performance gap is grew up during the years. This difference has led developers to exploit the GPUs to execute the most computationally intensive parts of their programs. The reason of this gap is the different architectures that these devices present (Figure 1.5). The CPU architecture maximizes the sequential programs performance. It is characterized by a refined control unit which allows to execute in parallel the instructions of a specific thread by maintaining a sequential aspect. The big *cache* memories allow to reduce the latency accessing to data and instructions memories. Another difference is the backward compatibility which is crucial in the CPUs since they have to be compliant with several

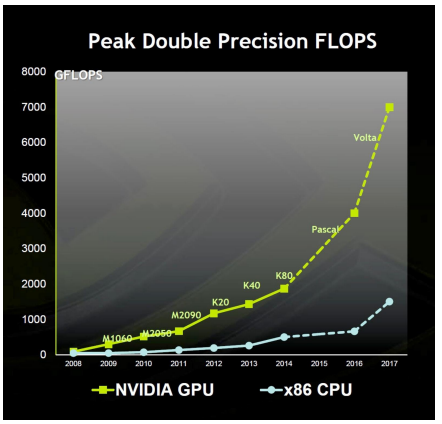


Figure 1.4: Peak double precision FLOPS. The graph shows the performances of the GPU and CPU architectures. From 2014 the two lines are dotted because they are trend previsions.

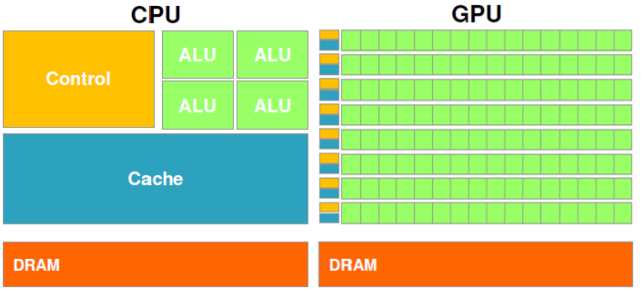


Figure 1.5: CPU and GPU architectures [20].

operative systems, applications and I/O systems. GPUs do not have to satisfy this requirement so restrictively, in fact their memory model is simpler. For this reason, the GPUs developer could increase the bandwith making these devices faster in the memory comunication than the CPUs. The most of the chip area is dedicated to the ALU to maximize the computational throughput. This make the control unit and the cache memories smaller

than the CPU ones (Figure 1.5). This architecture allows some threads to elaborate data while others are performing memory accesses. Furthermore, the GPUs are characterized by several small caches that allow to increase the memory bandwidth. Despite the GPUs are exploited to optimized the computation performances, there are some tasks that are more efficient if run by the CPU. For this reason, the most of applications uses the CPU for the sequential part of the code and the GPUs for the most intensive parts. Moreover, it is possible to exploit the parallelism in both devices using, on the CPU, the OpenMP paradigm for the parallel parts which involve a small amount of data.
In November 2006 NVIDIA presented a new typology of GPU based on G80 architecture which unified the graphic with the general purpose computing. From that moment, NVIDIA invested in the architectures development producing about a new version per year. The general philosophy of the different GPUs generations is to equip the devices with hundreds of cores organized in the so-called *streaming multiprocessors* (SMs). All the cores of the SM share the on-chip memory, called *shared memory*, the registers and spe-

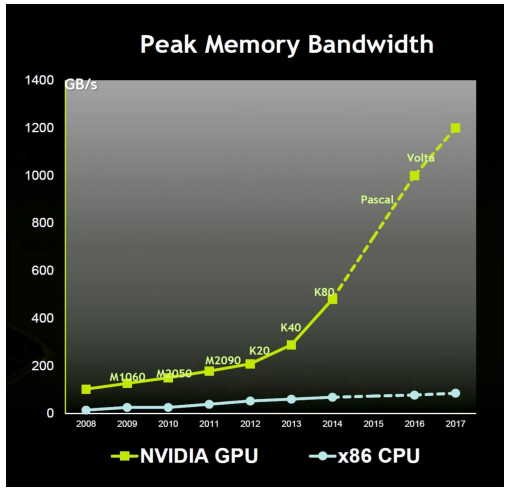


Figure 1.6: Peak memory bandwidth. The graph shows the memory bandwidth of the GPU and CPU devices. From 2014 the two lines are dotted because they are trend previsions.

Figure 1.7: Streaming Multiprocessor scheme in the Kepler architecture [21].

cial purpose resources such as the *Special Function Units* (SFUs) which perform trascendental instructions (Figure 1.7). Figure 1.8 shows the entire architecture of the Kepler version. The GPU architecture is built by adding several SMs on the same chip area. The presented Kepler GPU is equipped with 15 SMs and is connected to the CPU through PCI Express. GPU architectures are characterized by two types of memories. The *global memory* can be accessed by a thread independently from its block. It is used to exchange data between host and device through suitable CUDA functions. Since it takes long time to access this memory, due to the low access bandwidth, an efficient strategy has to be developed during the data transfers between GPU and CPU. The *on-chip* memories, shared memory and registers, have a low access latency and a high access bandwidth. In particular, registers are private to each threads while shared memory are private to all the threads of a block. The usage of the on-chip memories allows to reduce the high cost of accessing the global memory.

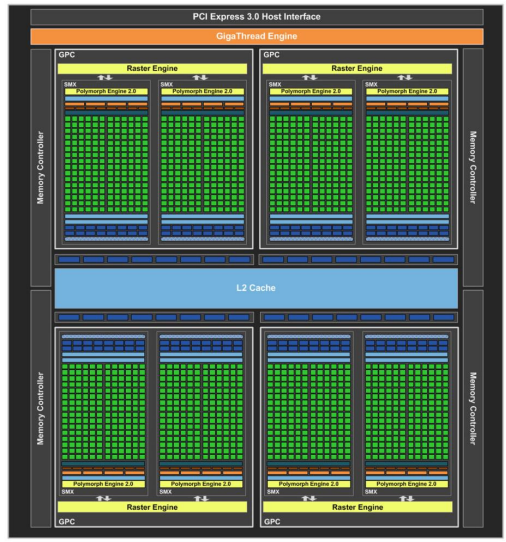


Figure 1.8: Kepler architecture scheme [21].

Compute Device Unified Architecture

Compute Device Unified Architecture (CUDA) is a parallel computing platform and programming model developed by NVIDIA for general computing on GPUs [19]. The system for the CUDA programmer is made up of an *host*, typically a CPU, interacting with one or more *devices* (the GPUs) used to speed up the computationally intensive parts of the code. The source code of a CUDA program contains both the parts related to the host and the devices, which are written in the same language (C/C++ or Fortran). Furthermore, the device parts are extended with the CUDA functions and instructions to be executed on the GPU. Each function performed by parallel threads on the device is called *kernel*. The CUDA program execution starts from the host that will invoke a kernel. At this point, the device generates a great number of threads to exploit the parallelism. Threads are organized in *blocks* which, in turn, constitute a *grid*. When all the threads in a grid end the execution, this continues on the host since

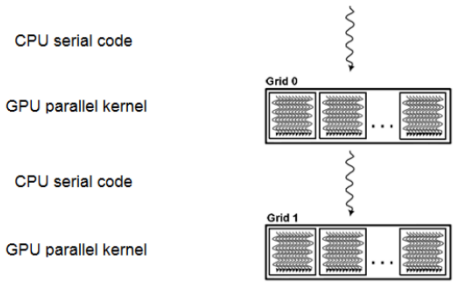


Figure 1.9: CUDA code execution [20].

another kernel is called. When a kernel ends it is important to transfer its results from the device to the host memory and to release the GPU memory. An example of CUDA code execution is shown in Figure 1.9. As already said, when a kernel is invoked, the execution passes from the host to the device. Here a series of tasks has to be performed. First of all, to allocate the memory on the device the function *cudaMalloc* is used by the host. This function has the following prototype:

cudaError_t cudaMalloc (void** devPtr, size_t size)

where *devPtr* is the pointer address and *size* the size in byte of the element to be allocated. *cudaError_t* is the function output that reports the presence of an error. Once allocated the memory, it is possible to transfer elements from host to device using the *cudaMemcpy* function:

cudaError_t cudaMemcpy (void* dst, const void* src, size_t count, enum CudaMemcpyKind kind)

where *dst* and *src* are the destination and the source memory addresses, respectively. *count* is the size in byte of the element to copy and *kind* defines the type of transfer, identifying the source and the destination. The

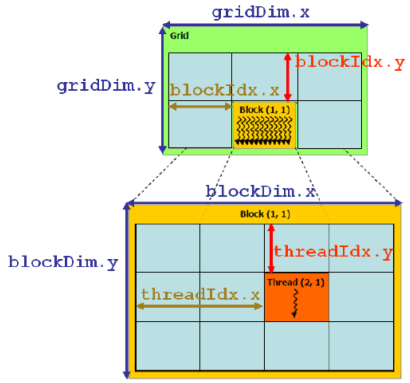


Figure 1.10: Threads, blocks and grid organization [20].

transfer direction can be host-host (*cudaMemcpyHostToHost*), host-device (*cudaMemcpyHostToDevice*, device-host (*cudaMemcpyDeviceToHost*) and device-device (*cudaMemcpyDeviceToDevice*). Typically when a kernel is invoked data are copied from host to device and, at the end of the kernel computation, data are copied from device to host. Once the device is initialized, the number of threads and blocks of the kernel have to be defined. CUDA provides variables to uniquely identify a block inside a grid using three-dimensional coordinates: *blockIdx.x*, *blockIdx.y* and *blockIdx.z*. In the same way a thread can be defined in a block using the *threadIdx.x*, *threadIdx.y* and *threadIdx.z* variables. It is possible to use only two dimensions in the thread identification, as shown in Figure 1.10. Furthermore, it is mandatory that all that blocks have the same number of threads. Finally, the variables *blockDim* and *gridDim* represent the blocks and grid dimension. In the two-dimensional case, the single thread can be identified as follows:

$$i = blockIdx.y * blockDim.y + threadIdex.y \tag{1.1}$$

$$j = blockIdx.x * blockDim.x + threadIdex.x \tag{1.2}$$

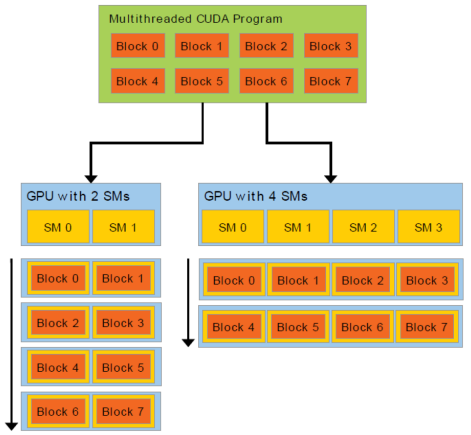


Figure 1.11: Automatic scalability [22].

If only one index has to be used to define a thread, it is computed as:

$$index = i * gridDim.x * blockDim.x + j \tag{1.3}$$

Moreover, it is possible to synchronize the activity of all the threads in a block using the *_syncthreads* function. In this way, all the threads that reach the barrier wait the others inside the block before continuing the execution. Since the synchronization has effect only inside the block, the system can execute the blocks in a random order. For example, Figure 1.11 shows how the same code can be executed in different ways in two boards equipped with a different number of cores. This advantage is called *automatic scalability*.

At this point of the program execution on the device, it is possible to invoke the kernel specifying the grid and the block dimensions, indicated with *grid* and *block* in the following:

kernel <<<dim3 grid, dim3 block>>>(arg1, arg2, ...)

where *arg1* and *arg2* are the function parameters. Once a kernel is invoked, each SM can manage up to 16 blocks on the base of its resources. In order

to be executed, each block assigned to a SM is split into groups of 32 consecutive threads, called *warps*.
Once the kernel execution is completed, the result can be transferred from device to host exploit *cudaMemcpy* function presented above, where *kind* is *cudaMemcpyDeviceToHost*. At the end of the device code execution, the GPU memory has to be released invoking the *cudaFree* function:

```
cudaError_t cudaFree(void* devPtr)
```

where *devPtr* is the device memory pointer to deallocate.
Once the host invokes a kernel, it can wait the results from the device and only then continues the program execution, or it can continue the execution immediately after the GPU function call. In the first case, there is a synchronization between the host and the device activity while, in the second, the two executions are asynchronous. In this last case, the kernel launches can be overlapped with the host function calls. Since the *cudaMemcpy* is a synchronous function, CUDA provides a tool to exploit the *concurrency* that is the ability to perform the CUDA kernel, the memory transfers and the host operations simultaneously. In this case, the function used to asynchronously transfer data from host to device, and viceversa, is the *cudaMemcpyAsync*. This function requires that the host memory is allocated through the *cudaMallocHost* function. In order to perform concurrency, CUDA provides the *streams* which allow CUDA operations, assigned to different streams, to be executed concurrently and be overlapped. For example, it is possible to overlap memory transfers with device or host execution, increasing the performances. Whenever the host, during its execution, needs the GPU computation results a *cudaDeviceSynchronize* function has to be called in order to block the host until the CUDA kernel is completed and the results transferred. Figure 1.12 shows a *Serial* execution where a memory transfer from host to device, a kernel and another memory transfer from device to host are sequentially performed. Using the streams it is possible to overlap transfers and kernel execution.

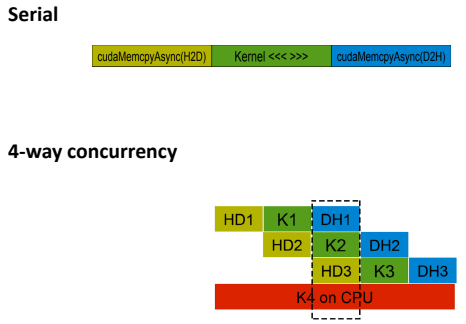


Figure 1.12: Stream usage. In the serial execution, memory transfers and kernel are computed sequentially. Using the streams it is possible to overlap these tasks [23].

CUDA Program Compilation

To compile a C-CUDA code, CUDA provides the *nvcc* compiler. In Figure 1.13 it is possible to see how this compiler works. The host code is compiled by a standard compiler such as *gcc* or *cl* while the device code is assigned to assembly form called *Parallel Thread eXecution* (PTX) and/or binary form. To indicate the GPU architecture to the compiler, the parameter *compute capability* is set. It is made up of two numbers which indicate the architecture and the version, respectively. The compute capability values of the systems used in this work will be presented below.

NVIDIA GPUDirect

As previously said, several medical applications search solutions to complex scientific and mathematical problems that are heavy from the computational point of view. They require platforms that delivers the highest throughput and lowest latency possible. In these cases of intensive workload, the multi-GPU systems are used exploiting the GPUDirect [24], which allows devices to read and write CUDA host and device memory, avoiding

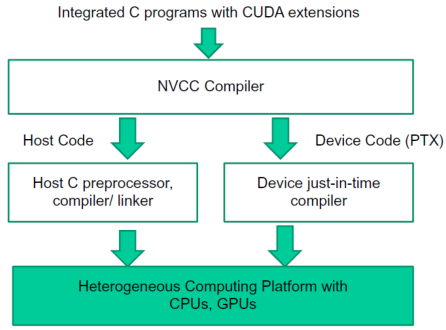


Figure 1.13: *nvcc* compiler [22].

unnecessary memory copies. In 2011 the GPUDirect Peer to Peer has been released: it allows to transfer data, to direct load and store access between GPUs on the same PCI Express root complex. Moreover, in 2013 the GPU Direct RDMA enables third party PCI Express devices to directly access to GPUs bypassing the CPU host memory, increasing the application performances. Figure 1.14 shows a direct Peer to Peer access and transfer between GPUs connected by the same PCI Express. In order to perform a peer to peer memory access, the GPU has to be characterized by a compute capability starting from 2.0. This access must be enabled between two devices by calling the *cudaDeviceEnablePeerAccess* function, as shown in this example [22]:

```
1 cudaSetDevice(0);
2 float* p0;
3 size_t size = 1024 * sizeof(float);
4 cudaMalloc(&p0, size);
5 MyKernel<<<1000, 128>>>(p0);
6 cudaSetDevice(1);
7 cudaDeviceEnablePeerAccess(0, 0);
8 MyKernel<<<1000, 128>>>(p0);
```

The first instruction sets the device 0 as the current device. In this device

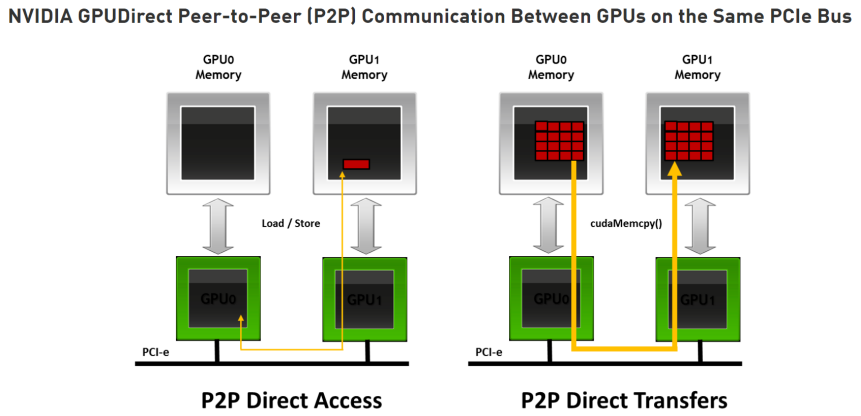


Figure 1.14: GPUDirect access and transfer [24].

the element *p0*, whose dimension is *size*, is allocated through the *cudaMalloc* function. Then *MyKernel* is invoked always in the same device. At this point, the *cudaSetDevice* sets the device 1 as current and the peer to peer access with device 0 is enabled. The kernel *MyKernel* is now launched on the device 1 and it can access the device 0 memory at address *p0*.
The memory copies can be performed between two different devices exploiting the *cudaMemcpyPeer* or the *cudaMemcpyPeerAsync* functions. The following code shows an example.

```
1  cudaSetDevice(0);
2  float* p0;
3  size_t size = 1024 * sizeof(float);
4  cudaMalloc(&p0, size);
5  cudaSetDevice(1);
6  float* p1;
7  cudaMalloc(&p1, size);
8  cudaSetDevice(0);
9  MyKernel<<<1000, 128>>>(p0);
10 cudaSetDevice(1);
11 cudaMemcpyPeer(p1, 1, p0, 0, size);
12 MyKernel<<<1000, 128>>>(p1);
```

In this case *p0* and *p1* are allocated on the device 0 and on the device 1, respectively. Once *MyKernel* is completed on the device 0, the execution passes on the device 1 where the *cudaMemcpyPeer* function is called to copy the element *p0* (from device 0 memory) to the element *p1* (in the device 1 memory). At the end, the last kernel is launched on the device 1. This copy is performed by an implicit NULL stream but if an asynchronous copy has to be executed, a stream has to be created on the device 1 and the function to call is:

```
1 ...
2 cudaSetDevice(1);
3 cudaMemcpyPeerAsync(p1, 1, p0, 0, size, mystream);
4 ...
```

Test systems

As for the applications presented in the following chapters, the tests are carried out on two different systems whose characteristics are shown in the Table 1.1.

The *System 1* is equipped with two NVIDIA Tesla K40 GPU, whose architecture is the Kepler (compute capability 3.5). Each board presents the features described in Table 1.1 [25]. This board is optimized for scientific computation therefore it does not present a graphical output. The second system (*System 2*) is equipped with a NVIDIA GTX1060 GPU which is based on the Pascal architecture (compute capability 6.0) [26]. This is a more recent GPU with a graphical output port.

	System 1		**System 2**	
	Intel i7 3770	**ITesla K40**	**Intel i7 6700**	**GTX 1060**
Cores number	4	2,880	4	1,152
RAM [GB]	8	12	32	3
Working frequency [GHz]	3.40	0.875	3.40	1.5

Table 1.1: Systems features.

Chapter 2

A realistic cerebellar granular layer simulator on parallel technologies

The knowledge of the physiological principles which underlie the brain function and that let the human being to learn, act and remember is one of the main challenges addressed to neuroscientists and engineers. The interest in this challenge is raised up with the ongoing technological innovation and neurological discoveries which make the scientist goals very ambitious. One of these goals is the development of a model capable of accurately simulating the brain behaviour. A model which reproduces the brain activity in a realistic way is heavy from the computational point of view if the different types of neurons and synapses and their complex dynamic properties are considered. Due to this computational complexity, the scientists need to perform their simulations exploiting the HPC technologies in order to decrease the computational times, trying to reach the real-time constraint.

2.1 Introduction

The interest in the understanding the human brain and its functionalities has been raised up significantly that the scientific community has promulgated several projects in order to promote the research in this field. In particular, the aim of the Human Brain Project (HBP) is to extend the brain knowledge to help the brain disorders diagnosis, to elaborate new therapies and to facilitate the design of new brain-inspired technologies [27]. HBP has been selected as European Commission Future and Emerging Technologies Flagship that involves a consortium of 112 partners and 24 European countries. It started in October 2013. HBP aims to create a wide collaboration between all the partners so that they can share their results in order to facilitate the research progress. In fact, one of the main problem in the neurophysiology field is the fragmentation of the brain research and the data that it produces [28]. For this reason, an integrated system of ICT-based research platforms has been developed in order to contain data belonging to several research groups and partners. Each group deals with the study and the understanding of different parts and aspects of the brain since another project goal is to reach a complete brain knowledge considering several levels of organisation, starting from the *protein* level to the *entire body* passing through the *chromosomes*, *synapses*, *cells*, *microcircuits*, *brain regions*, *brain* (Figure 2.1). Having a complete brain knowledge means that if a particular phenomenon occurs in one of the levels, it is possible to trace it back in the others in order to study its effects. A deep brain understanding and the development of new neuronal models with the ability to reproduce its complex function could have positive effects in different research fields: neuroscientists will be able to discover and study new mechanisms and behaviours, as well as conduct investigations that due to their invasiveness today are regarded as impossible. From the medical point of view, it will be possible to discover new diseases diagnosis and therapies. A brain simulator will allow to study innovative drug treatments and their effects on different brain organisation levels. Furthermore, a brain simulator will allow researchers to identificate the *biological signatures* of the neurological deseases promoting new in silico

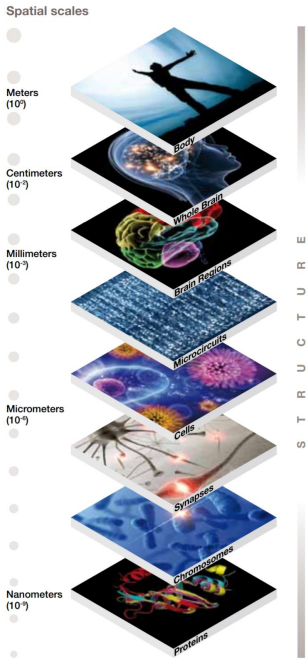


Figure 2.1: Different organisation levels in the brain study. The image shows different levels of organisation of the brain, covering nine orders of magnitude [28].

experiments to study the causes and effects of these pathologies. Finally, a deep brain understanding will promote the development of innovative technologies whose architectures will be inspired by the neurons elaboration, learning and cognitive capabilities and will achieve high-energy efficiency and fault tolerance [27]. The understanding of how the human brain works, the development and the simulation of complex neuronal models and the data collection demand more than a standard computer. This need of HPC technologies and, in particular, of supercomputers has meant that one of the six platforms of the HBP infrastructure (called *High Performance Analytics & Computing Platform*) is dedicated to provide some of the most

powerful computers in the world to scientists in order to facilitate their research. The other platforms are: *Neuroinformatics* which collects and integrates neuroscience data, *Medical Informatics* which collects medical data, *Neuromorphic Computing* for the hardware implementation of the brain functions, *Neurorobotics* which provides researchers with a virtual or real *body* in which to simulate the developed brain models and analyse the effects (i.e., the movement control), the stimulus reaction and how the robot learns in the virtual environment. Finally, the platform related to the development of the neuronal models based on the brain circuits and functions is called *Brain Simulation* [3]. The University of Pavia team is involved in this platform with the goal of developing a realistic cerebellum simulator. A part of this group works in the development of the mathematical models of different cerebellar neurons and, then, of the network that integrates these cells. Once the models are completed, another part of the group analyses them and studies an efficient way to exploit them on high performance computing technologies. The same work flow has been followed in the next paragraphs: at first a mention of the cerebellum physiology is provided. Then, the mathematical models of the cerebellar cells are presented in order to better understand the algorithms developed to reproduce the neuronal behaviours. The first aim of the work has been to evaluate the algorithms perfomances in order to study and develop the most efficient version for each cell. The second part of this chapter is related to the design and the simulation of the *granular layer* of the cerebellar cortex exploiting HPC technologies.

2.2 Physiology of the cerebellar cortex

2.2.1 The cerebellum and the cerebellar cortex

The cerebellum is the part of the brain whose main function is to calibrate and control the voluntary and involuntary actions of the human body. Several parts of the encephalon and of the sense organs send to the cerebellum signals concerning the planned movements. The cerebellum is able to compare these different signals and to regulate the strengh and the direc-

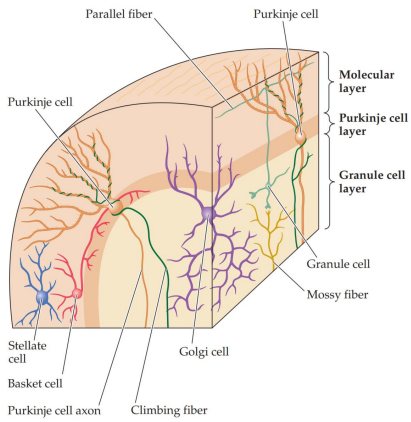


Figure 2.2: Cerebellar cortex. Different types of neurons in the cerebellar cortex [29].

tions of the body movements whenever they are different from the planned actions. It can remember the movements and correct the ones performed in an incorrect way. It has a main role in the correct posture and equilibrium control. This work is focused on the function of the *cerebellar cortex* which covers the cerebellum surface. It is made up of five types of neurons (*Purkinje*, *granular*, *Golgi*, *stellate* and *basket cells*) which are displaced in three different layers: the *granular layer*, the *Purkinje layer* and the *molecular layer*. The neurons dendrites and axons are linked through the *excitatory* and *inhibitory* synapses, creating a circuit (Figure 2.2) [29]. The *mossy fibers* are one of the cerebellar cortex inputs. They come from the *pontine nuclei* and synapse on the granule cells which are in the granular layer. This is the largest type of neurons in the human brain and the signals that they generate go through the Purkinje layer and end in the molecular one, where the dendritic trees of the Purkinje cells are displaced. The granule axon has a particular shape since, when it reaches the molecular layer, it forms a T-shaped branches in order to relay its excitatory signal into the Purkinje dendrites. In fact, the Purkinje cells have their body in the Purkinje layer but, as previously said, their dendritic trees are in the molecular layer. The Purkinje cells also receive the signals of the inferior olive through the *climb-*

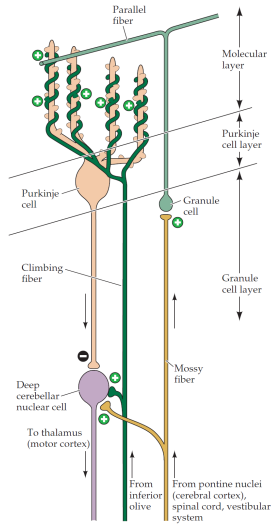


Figure 2.3: Functional module of the cerebellar cortex. Excitatory and inhibitory connections between the Purkinje, Golgi and granular cells [29].

ing fibers, while their output reaches the deep cerebellar nuclei, as shown in Figure 2.3. This is the only output of the cerebellar cortex and it consists in an inhibitory signal. The deep cerebellar nuclei receive also excitatory inputs from the mossy and climbing fibers. The molecular layer contains also the dendritic trees of the Golgi cells, whose body is placed in the granular layer. These cells receive the inputs from the granular cells (through their ascending axon and parallel fibers) and provide an inhibitory output, to the granules, which may limit the granule excitatory inputs to the Purkinje cells. This circuit, illustrated in Figure 2.3, is the main function module of the cerebellum where it is repeated several times. The modulation of the signals through these circuits provides the real-time control of the movements that the body performs and it underlies the motor learning taking into account the long-term changes [29].

2.2.2 The action potential

Neurons can communicate with each other through the exchange of electrical signals whose generation and trasmission are due to the *selective permeability* of their membranes to several types of ions. The permeability of the membrane is due to the presence of the *ion channels* which allow specific ions (Na^+, Ca^{2+}, K^+ and Cl^-) to cross it in the direction of their concentration gradient [29]. In fact, the membrane is characterized by different concentrations of specific iones on its sides. These properties allow the genesis of electrical signals called *action potentials* or *spikes*. At rest, the membrane is characterized by a *resting membrane potential*, typically from -40 to -90 mV on the base of the neuron typology. The action potential can be generated when a current crosses the membrane modifying its resting potential. In normal conditions, this stimulus is due to the synaptic activity, which allows neurons to communicate, while in the laboratory it can be generated connecting a microelectrode to a battery. There is an *hyperpolarization* if the injected current decreases the potential values. Otherwise, if the injected current increases the resting membrane potential values there is a *depolarization*. If these values overcome a *threshold potential* an action potential occurs. The spike is the neuron answer to a stimulus and it is a fast (about 1 ms) change from negative to positive in the transmembrane potential [29]. The spike amplitude is not proportional to the injected current intensity used to evoke it: this is a *all-or-none* phenomenon where the action potential fully occurs or not at all. Figure 2.4.A shows two microelectrodes introduced in a neuron body to stimulate the cell and record the membrane potential variation. Figure 2.4.B presents these potential variations due to different current stimuli. Analyzing the Figure 2.4.B it can be noticed that when a current is not injected, the membrane potential reaches the resting value. When a negative current is injected, the potential decreases while, when the current is positive, the potential can increase its value staying under the threshold or overcoming the threshold and generating the action potential. If the current amplitude or duration are increased, multiple action potentials occur reaching always the same maximum value ($\sim$40 mV).

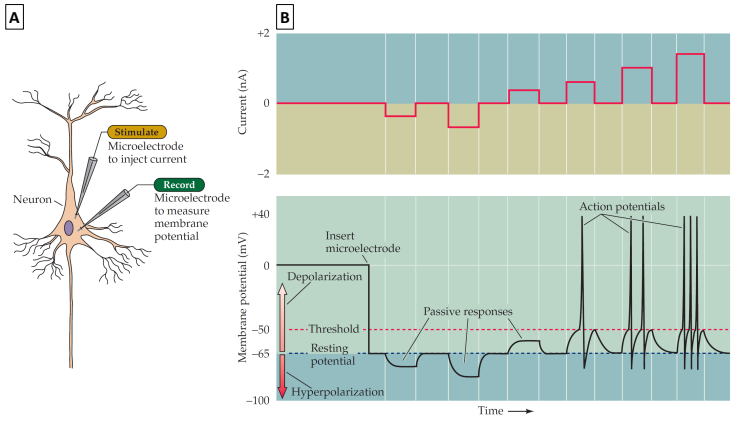


Figure 2.4: Action potentials in a neuronal cell. A) Two micrelectrodes are introduced in the neuron body: the former to stimulate it, the latter to record the membrane potential variations. B) The membrane potential variation due to different current stimuli [29].

At rest, i.e. in a condition of electrochemical equilibrium, the value of the membrane potential is called *equilibrium potential* computed through the *Nernst equation* (Equation 2.1):

$$E_x = \frac{RT}{zF} ln \frac{[X]_e}{[X]_i} \tag{2.1}$$

where E_x is the equilibrium potential for any ion X, R is the gas constant, T is the absolute temperature (in degrees on the Kelvin scale), z is the valence of the permeant ion and F is the Faraday constant [29]. $[X]_e$ and $[X]_i$ are the ionic concentrations at the two sides of the membrane. The *Goldman equation* (Equation 2.2) computes the membrane potential taking into account the permeabilities of the ions that are able to cross the membrane.

$$V_m = \frac{RT}{zF} ln \frac{P_K[K^+]_e + P_{Na}[Na^+]_e + P_{Cl}[Cl^-]_i}{P_K[K^+]_i + P_{Na}[Na^+]_i + P_{Cl}[Cl^-]_e} \tag{2.2}$$

As the permeability of the ion (P_x) changes, also the membrane potential varies and, if it reaches the threshold, an action potential occurs. The potas-

sium permeability (P_K) determines the resting membrane potential, in fact these ionic channels are opened at rest condition. The sodium permeability increases when there is a depolarization which reaches the threshold. In this case, the sodium ions enter in the neuron, further increasing the potential value which reaches the E_{Na} potential. Then, the sodium channels are inactivated while the potassium ones are slowly activated, allowing the potassium leakage outside the membrane. At this point the potential decreases (*repolarization*) until it reaches the E_K value. A short refractory period occurs where the potential decreases more than E_K and the neuron is insensitive to the stimuli. At the end, the hyperpolarization closes the potassium channels and the potential reaches the resting value [29].

2.2.3 The synapse and the receptors

The communication between neurons, and then the signal trasmission, is performed in the *synapses*, which are elements that link the axon of the *presynaptic* neuron to the dendrite or the body of the *postsynaptic* one. The mechanism that underlies the signals trasmission defines the type of synapse that can be *electrical* or *chemical*. The former has the presynaptic and postsynaptic membranes very close and linked through the *gap juctions* where the current can flow. The latter (Figure 2.5) has the presynaptic neuron separated from the postsynaptic one by a *synaptic cleft* where the signal trasmission is performed by the *neurotransmitters* contained in the *synaptic vesicles* (Figure 2.5.1) in the presynaptic neuron. When the presynaptic neuron is stimulated by an action potential (Figure 2.5.2), the membrane potential increases and causing voltage-gated Ca^{2+} channels opening (Figure 2.5.3). A Ca^{2+} ions flow inward is generated (Figure 2.5.4). The increased amount of Ca^{2+} ions in the presynaptic terminal causes the fusion of the membrane with the vesicles (Figure 2.5.5) and the neurotransmitter molecules release in the synaptic cleft (Figure 2.5.6). These molecules reach and bind with receptors displaced in the postsynaptic membrane (Figure 2.5.7). This connection causes the conformation of the receptors ionic channels that can turn in a *open*, *close* or *desensitized* state. If the channels are open (Figure 2.5.8), the ions start flowing through the ionic channels gener-

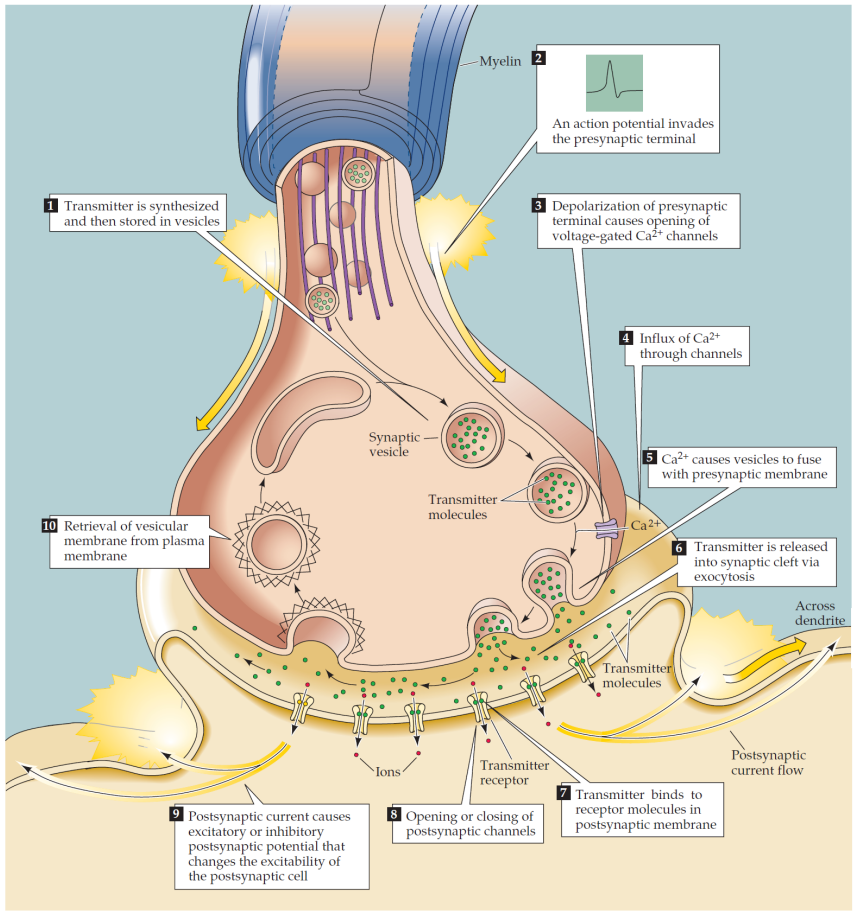


Figure 2.5: Chemical synapse. Events that occur in a chemical synapse when a signal is transmitted [29].

ating ionic flows that can depolarize or hyperpolarize the postsynaptic neuron potential (Figure 2.5.9): in the first case, the postsynaptic potential is *excitatory* while in the second case it is *inhibitory*. At the end, the retrieval of the vesicular membrane from the plasma membrane is performed (Figure

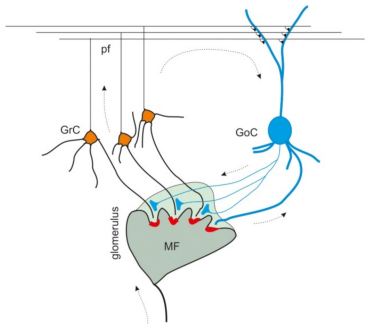


Figure 2.6: Chemical synapses in the granuar layer of the cerebellar cortex. Golgi cells, granular cells and mossy fibers make synapses in the glomerulus. [30].

2.5.10). The most common neurotransmitter in the cerebellum is the glutammate and the main receptors of this molecule are the NMDA, activated by the molecule *N-Methyl-D-aspartate* (NMDA), and the AMPA, whose agonist molecule is the α-amino-3-hydroxy-5-methyl-4-isoxazolepropionic acid (AMPA). Most of the neurons are characterized by both these receptors which allow the flow of the sodium, potassium (and also calcium for the NMDA) ions. These receptors cause the excitatory postsynaptic potentials with a different timing: the NMDA receptor is slower since at first its channel is blocked by the magnesium (Mg^{2+}) ion. The principal inhibitory receptor is the *gamma-Aminobutyric acid* (GABA) which is made up of two different subunits and allows the chloride (Cl^-) ions to flow inside the membrane causing the neuron inhibition [29].
Concerning the cerebellar cortex, Figure 2.6 shows (in red) the excitatory synapses between mossy fibers and granular cells and (in light blue) the inhibitory synapses between Golgi and granule cells. These synapses are performed in a particular structure, called *glomerulus*, which hosts the Golgi cell axon that ihnibits the granular cells dendrites (through the GABAergic synapses) and the mossy fiber terminal which excites the granules dendrites (through the glutamatergic synapses) (Figure 2.6).

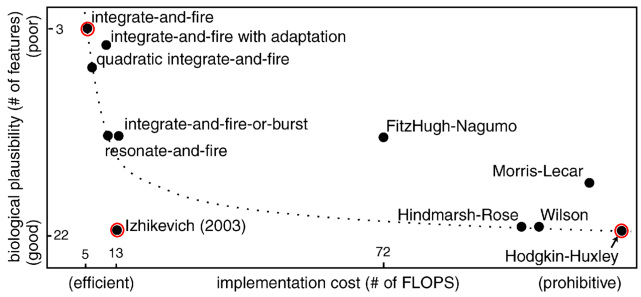


Figure 2.7: Different neuronal models. Each model is characterized by a biological plausibility and an implementational cost [31].

2.3 Neuronal models: state of the art

In literature, several neuronal models more or less realistic are present. Their *biological plausibility* is discussed by author of [31] on the base of the behaviors that they can reproduce. The more functionalities a model can reproduce the more it is complex. In Figure 2.7 different models are shown on the base of their biological plausibility (i.e. the number of features that they are able to replicate) and the implementation cost. As it will be discussed at the end of the paragraph, the ability of a model to reproduce a high amount of mechanisms implies a higher number of equations and so a higher computational complexity. In Figure 2.7 the red circles indicate some of the most used models which are explained in the following paragraphs.

2.3.1 Leaky Integrate-and-Fire model

The *Leaky Integrate-and Fire* (LIF) is one of the simplest and, for this reason, most used models in computational neuroscience. It is characterized by only one differential equation (Equation 2.3) which gives a simple relation between the membrane potential v, the synaptic current and, even-

tually, an injected current I.

$$C_m \frac{dv}{dt} = -v(t) + RI(t) \tag{2.3}$$

The equation is characterized by a capacitor C_m and a resistor R. When the membrane potential v reaches a threshold v_{th}, it is instantaneously set to the resting membrane potential v_r and it will mantain this value for a small period called *refractory period*. Then, a new membrane potential evaluation restarts as in Equation 2.3. This model is not able to reproduce complex and very important behaviors such as the *spike latency* and it has a fixed threshold. For these reasons, its biological plausibility is very low. Despite this, it is very fast if the computational time is considered. It is used in those applications that do not require a high level of realism but which need only to know if a neuron has generated a spike or not [31].

2.3.2 Izhikevich model

The Izhikevich model is more realistic than the LIF one and it is characterized by the following differential equations:

$$\begin{cases} \frac{dv}{dt} = 0.04v^2 + 5v + 140 - u + I \\ \frac{du}{dt} = a(bv - u) \end{cases}$$

where v is the potential membrane, the variable u considers the membrane recovery caused by the inactivation of Na^+ ionic current and the activation of K^+ ionic current, I is the total current and a and b are suitable constants. Once the spike has occurred, the model resets the variables v and u as described in the following system:

$$if\ v \geq 30\ mV\ then \begin{cases} v = c \\ u = u + d \end{cases}$$

where 30 mV is the peak value of the action potential, while c and d are suitable costants. This model increases the level of biological plausibility compared to the LIF since it has not a fixed threshold and it is able to

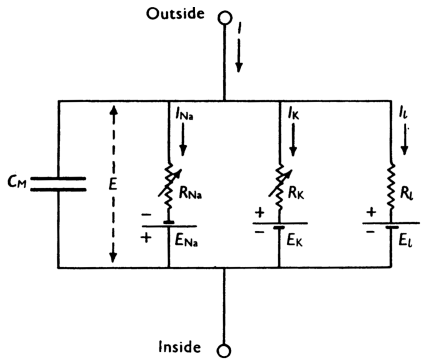


Figure 2.8: The Hodgkin and Huxley model. Circuital model of the cellular membrane [32].

reproduce different neurons behaviours. Nevertheless, it does not describe all the physiological mechanisms of a neuron because it considers only the membrane potential variations.

2.3.3 Hodgkin-Huxley model

The Hodgkin and Huxley (HH) model [32] is one of the most important models due to the physiological realism that it reproduces. It is able to describe the ionic channel dynamics and how the membrane permeability depends on the membrane potential. It allows the study of aspects related to the synaptic integration and it is able to exhibit the complete meaningful neuronal behavior [33]. Hodgkin and Huxley described the cellular membrane as a capacitor C_m since it keeps the ions separated on its sides (Figure 2.8). The resistors stand for the ionic channels, contained in the membrane, that allow the ions crossing. The voltage generators represent the active transport mechanisms that characterized the cellular activity. The current I that flows through the membrane is described as (Equation 2.4):

$$I = C_m \frac{dV_m}{dt} + I_{ion} \tag{2.4}$$

where V_m is the membrane potential and I_{ion} is the sum of the ionic currents. In this model the Na^+ and the K^+ channels dynamics are described. Furthermore, a leakage current I_L that represents the ionic fluxes in always-opened channels is included. For these reasons the term I_{ion} in Equation 2.4 is the sum of these three contributions (Equation 2.5):

$$I_{ion} = I_{Na} + I_K + I_L \tag{2.5}$$

Each ionic current is defined as the product between the channel conductance g_{ion} and the difference between the membrane potential V_m and the equilibrium potential of the specific ion E_{ion}, as shown in Equation 2.6.

$$I_{ion} = g_{ion}(V_m - E_{ion}) \tag{2.6}$$

The ionic channels are characterized by the presence of particles, called *gating particles*, whose position allows the channel opening or closure. The HH model can reproduce their dynamic computing the channel conductance as follows (Equation 2.7):

$$g_{ion} = \overline{g}_{ion} * x_{ion}^z * y_{ion}^k \tag{2.7}$$

where $\overline{g}_{\text{ion}}$ is the maximum conductance of the channel, x_{ion} and y_{ion} are the state variables of the gating particles. They are the probabilities that the gating particles occupy a certain position in the membrane. z and k represent the number of activation and inactivation particles for each channel [33]. The sodium channel has three activation particles m and one inactivation particle h while the potassium channel has four gating particles n. Their conductances are computed as in Equation 2.8 and Equation 2.9.

$$g_{Na} = \overline{g}_{Na} * m^3 * h \tag{2.8}$$

$$g_K = \overline{g}_K * n^4 \tag{2.9}$$

If the potassium conductance is considered, the variation of the gating particle n over the time is described by the following first-order reaction between open and closed state:

$$\text{(1-n)} \underset{\beta_{\text{n}}}{\overset{\alpha_{\text{n}}}{\rightleftharpoons}} \text{n}$$

where α_n and β_n are rate variables which depend on the potential. This reaction is described by the Equation 2.10:

$$\frac{dn}{dt} = \alpha_n(1-n) - \beta_n n \tag{2.10}$$

Assuming that:

$$n_\infty = \frac{\alpha_n}{\alpha_n + \beta_n} \tag{2.11}$$

$$\tau_n = \frac{1}{\alpha_n + \beta_n} \tag{2.12}$$

it is possible to rewrite Equation 2.10 as in Equation 2.13:

$$n(t) = n_\infty - (n_\infty - n) \cdot e^{-\frac{t}{\tau_n}} \tag{2.13}$$

The same considerations can be done for the sodium channel and for its gating particles m and h. Finally, after the gating particles definition, it is possible to conclude that the Equation 2.4 can be rewritten as follows:

$$I = C_m \frac{dV_m}{dt} + \overline{g}_K * n^4(V_m - V_K) + \overline{g}_{Na} * m^3 * h(V_m - V_{Na}) + \overline{g}_L(V_m - V_L) \tag{2.14}$$

As explained in this paragraph, the HH model is characterized by a higher amount of differential equations than the previous models. This characteristic allows, on one side, to reproduce the neuronal behavior with a high level of realism. On the other side, it increases the implementation cost and the simulation time. The choice of the model to use depends on the final application. In this work, a more complex version of the HH model is chosen since the goal is to reach a level of realism as high as possible. The inevitable consequence of the increased computational cost is managed exploiting the HPC technologies in order to reduce the execution time.

2.4 Neuronal models of the granule and the Golgi cell

The models used to reproduce the behavior of the Golgi and the granular soma are based on the HH but they present a higher number of ionic

channels. In Figure 2.8 it is possible to notice that each ionic channel is represented by a branch of the circuit characterized by a resistor and a voltage generator in series. In this new model a single channel can be represented by several branches in parallel since each branch reproduces a specific channel dynamic.

2.4.1 The granular cell model

Concerning the soma of the granular cell, the model has been improved, as described in [34], to take into account some particular mechanisms related to ions. The sodium channel, for instance, is represented by three currents: a fast Na^+ current ($I_{Na\text{-}f}$), a persistent Na^+ current ($I_{Na\text{-}p}$) and a resurgent Na^+ current ($I_{Na\text{-}r}$). The potassium channel is represented by five currents which reproduce different dynamics: a current for rectified delayed channels ($I_{K\text{-}V}$), a current depending on intracellular calcium concentration ($I_{K\text{-}Ca}$), a current for inward rectified channels ($I_{K\text{-}IR}$), a current for type-A channels ($I_{K\text{-}A}$) and a current for slow kinetic channels ($I_{K\text{-}slow}$). The Nernst potential of the sodium and of the potassium channels are constant during the neuronal activity evaluation. The calcium channel present in the granule is characterized by a variable intracellular calcium concentration [27]. The Ca^{2+} dynamic is described by the following differential equation (Equation 2.15):

$$\frac{d[Ca^{2+}]}{dt} = \frac{-I_{Ca}}{2FAd} - (\beta_{Ca}([Ca^{2+}] - [Ca^{2+}]_0)) \tag{2.15}$$

where d is the depth of the vesicle linked to the cellular membrane, whose area is indicated with A. β_{Ca} determines the calcium ions leakage from the cell, F is the Faraday costant, $[Ca^{2+}]_0$ is the calcium concentration at rest. Equation 2.15 allows to compute the updated $[Ca^{2+}]$ value and to evaluate a new Nernst potential E_{Ca} as described in Equation 2.1. The kinetic of these ionic channels is described using the HH model and the gating particle mechanism described above. Each channel is characterized by a different number of activation and inactivation particles.
In this channel dynamic the temperature has an important role since it de-

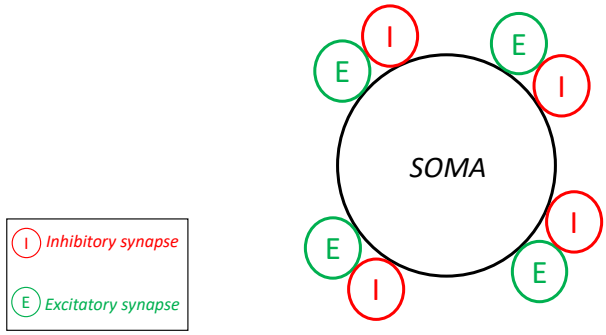


Figure 2.9: Granular cell representation. The model of the granule considers the cellular soma and four excitatory and four inhibitory synapses.

termines the speed at which the particles modify the channel conformation. For this reason, the variable $Q_{\Delta T}$ is considered in the simulation and it is computed as follows:

$$Q_{\Delta T} = Q_{10}^{\frac{T_{sim}-T_{esp}}{10}} \tag{2.16}$$

where T_{sim} is the temperature set in the simulation and T_{esp} is the one of the reference experiment [34]. The granule cell activity is reproduced by a mono-compartmental model (Figure 2.9) which provides a cellular soma reproduction through the ionic channels dynamics and the equations presented in this paragraph. The granules communicate with the other cells through the synapses that, in this work, are directly linked to the cellular soma, without taking into account axons and dendrites. Despite this approximation, this model is able to reproduce the cell behavior in a realistic way, as discussed in the following paragraphs. A typical behavior of the granule cell is shown in Figure 2.10 where the soma is stimulated with three different current injections: the first (10 pA) causes a potential increase but its value does not reach the threshold and, for this reason, the spikes do not occur. The next current injections (16 and 22 pA) excite the cell enough to reach potential values over the threshold, so the action potential are generated. It is important to notice that as far as the current

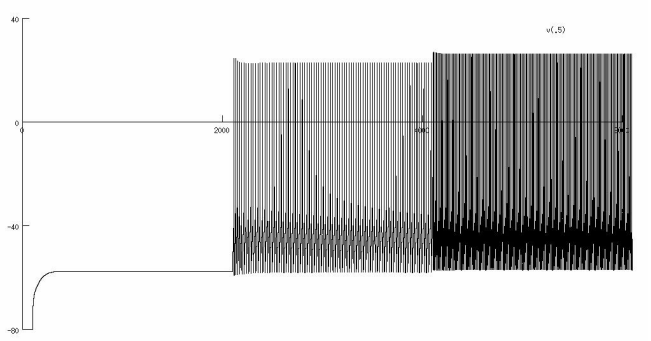


Figure 2.10: Granular cell behaviors. Three different current injection excite the granular cell. Graph generated with the NEURON simulator. On the x-axis the time expressed in ms and on the y-axis the potential is expressed in mV.

amplitude increases the spike frequency grows up, as already explained in Paragraph 2.2.2. According to the registrations reported in [35], at rest the granular cells do not show a *background* activity, so they generate spikes only if stimulated. The graph in Figure 2.10 is generated with the NEURON simulator [36] which is a typical software used for the neuron reconstruction and simulation and whose results are considered as golden reference for this work.

2.4.2 The Golgi cell model

Golgi cells are inhibitory inter-neurons that can elaborate and transmit signals in the granular layer of the cerebellum [33]. Unlike the granules, at rest Golgi cells show a background activity generating action potentials with a frequency of 6 Hz [35]. This normal low-frequency pacemaking, that the Golgi cell shows when it is not stimulated, is represented in Figure 2.11.A. When the cell is stimulated by a current, it increases the spike frequency as shown in Figure 2.11.B. The Golgi cell shows also two further behaviors. For example, if it is stimulated by a -180 pA current, it stops

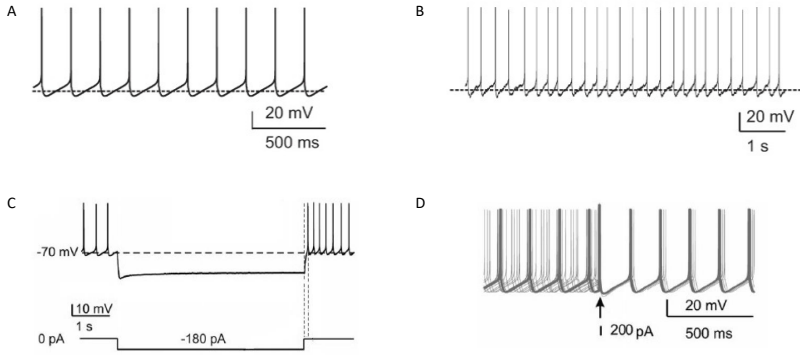


Figure 2.11: Golgi cell behaviors. A) Low frequency pacemaking; B) High frequency spike discharge after a current injection; C) Rebound excitation after a current injection of -180 pA; D) Phase reset of 6 Golgi cells after a current injection of 200 pA [33].

firing and, after a pause, it restarts with higher frequency. This behavior is called *rebound excitation* and it is shown in Figure 2.11.C. Finally, in Figure 2.11.D the *phase reset* is presented. An input of 200 pA allows six Golgi cells to restart spiking in phase without taking into account the preceding phase [33].
Also for the Golgi cell, the model adopted to reproduce its activity is an extended HH model version. The ionic currents that characterized the soma, in this model, and that can reproduce the regular pacemaking of the cell (Figure 2.11.A) are the sodium persistent current ($I_{Na\text{-}p}$), the h current (I_h), the SK-type small conductance calcium-dependent potassium current ($I_{K\text{-}AHP}$) and the slow M-like potassium current ($I_{K\text{-}SLOW}$). These currents together with the sodium resurgent current ($I_{Na\text{-}r}$) and the A-current ($I_{K\text{-}A}$) regulate the response frequency and delay [37]. The action potential generation is also due to the sodium transient current ($I_{Na\text{-}t}$) and K^+ current ($I_{K\text{-}V}$). In this model also three currents which depend to the Ca^{2+} intracellular concentration are considered: $I_{K\text{-}C}$ and $I_{Ca\text{-}HVA}$ are involved in

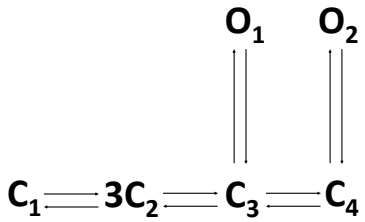


Figure 2.12: Markov gating scheme of the SK-type calcium dependent potassium channel [33].

the fast phase of a spike, which is present after the hyperpolarization, and the $I_{Ca\text{-}LVA}$ improves the rebound depolarization [33]. The rebound excitation, shown in Figure 2.11.C, is caused by the currents $I_{Ca\text{-}LVA}$ and I_h. All these currents are described by the gating particles model explained before. Instead, the $I_{K\text{-}AHP}$ current is simulated with a Markov gating scheme characterized by six states: C_1, C_2, C_3 and C_4 are the *closed* states while O_1 and O_2 are the *open* states (Figure 2.12). In this case the ionic conductance is computed considering the sum of the open states, as shown in Equation 2.17:

$$g_{K-AHP} = \overline{g}_{K-AHP} * (O_1 + O_2) \tag{2.17}$$

where $\overline{g}_{K\text{-}AHP}$ is the maximum conductance of the channel. The Golgi cell is reproduced by a multi-compartment model which considers the soma, dendrites and axon (Figure 2.13). The soma (in red in Figure 2.13) is a sphere and contains all the ionic channels shown before. Dendrites (in green) and axon (in blue) lack channels and consist of resistors providing connections between adjacent compartments. For this reason, they show passive properties. Soma is also connected to a micropipette for signals recording [33] [37].

2.5 Neuronal models of the chemical synapse

The granules and the Golgi cells can be stimulated by a current injection in the soma or through the synaptic current generated in the neurons com-

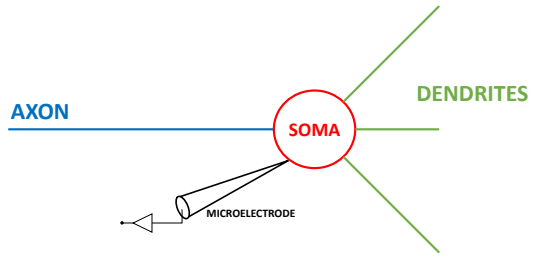


Figure 2.13: Multi-compartment model.

munication. To compute the synaptic current it is important to provide a model that reproduces the presynaptic and the postsynaptic dynamics, already described in Paragraph 2.2.3. The presynaptic dynamic is reproduced with a three-state kinetic scheme [38] where X is the amount of neurotransmitter available for the release (which is included in the vesicles), Y is the amount of released neurotransmitter and Z is the neurotransmitter recovered in the presynaptic terminal. The transactions between the states are defined by first-order kinetic reactions which are described by the following differential equation system:

$$\begin{cases} \frac{dX}{dt} = \frac{Z}{\tau_R} - P * X * \delta(t - t_{spike}) \\ \frac{dY}{dt} = -\frac{Y}{\tau_1} + P * X * \delta(t - t_{spike}) \\ \frac{dZ}{dt} = \frac{Y}{\tau_1} - \frac{Z}{\tau_R} \\ \frac{dP}{dt} = -\frac{P}{\tau_F} + p(1 - P) * \delta(t - t_{spike}) \end{cases}$$

where τ_R is a constant which describes the neurotransmitter recovery time, τ_1 is the inactivation constant, τ_F is the constant of the *synaptic facilitation*, which occurs when two or more potentials reach the presynaptic terminal in a very short range of time generating a transient increase of the synaptic strenght [29]. P is the release probability, p its initial value and δ the Dirac delta function. The excitatory and inhibitory action potentials occur in the

presynaptic terminal with different frequencies depending on the neuron typology. As the presynaptic terminals are activated, the neurotransmitter is released and this amount is given by the sum of two terms: an impulsive and a diffusive term (Equation 2.18).

$$[T] = [T]_P + [T]_D \tag{2.18}$$

where $[T]$ is the total amount of the released neurotransmitter, $[T]_P$ is the synaptic impulse and $[T]_D$ is the diffusive component which follows the impulsive one. In particular, the release of neurotransmitter represented with $[T]_P$ reaches the AMPA and the subunit α_1 of the GABA-A receptors, which are located in front of the synaptic cleft, so in front of the released site. The neurotransmitter molecules described by the diffusive component reach the NMDA and the subunit α_6 of the GABA-A receptors. The glutammate and GABA concentrations depend on the distance r from the released site. They are computed as follows:

$$[Glut]_d = \frac{M}{4h\pi D_{eff}t} e^{\frac{-r^2}{4D_{eff}t}} \tag{2.19}$$

$$[GABA]_d = \frac{M}{4h\pi D_{eff}t} e^{\frac{-r^2}{4D_{eff}t}} \tag{2.20}$$

where M is the number of released molecules, D_{eff} is the effective diffusion coefficient and h is the vescicle depth [38]. The link between the neurotransmitter and the postsynaptic receptors activates the kinetic scheme which are characterized by first-order reactions. The current which flows through the receptors channels is computed as in Equation 2.21.

$$I_{receptor} = \overline{g}_{receptor} * (V_m - V_{rev,receptor}) * O(T) \tag{2.21}$$

where $\overline{g}_{receptor}$ is the maximum conductance of the receptor channel, V_m the membrane potential and $V_{rev,receptor}$ is the ionic reversal potential. $O(T)$ is the kinetic scheme open state which depends on the neurotransmitter concentration [38]. Each receptor current is considered in the Equation 2.4 to update the membrane potential value.

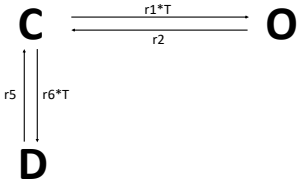


Figure 2.14: AMPA kinetic scheme. 3-state kinetic scheme: C, D and O are the *closed*, *open* and *desensitized* state respectively.

The kinetic scheme of the AMPA receptor (Figure 2.14) presents three states: the open state O, the closed state C and the desensitized state D. In order to take into account the temperature, all the kinetic costants (*r1*, *r2*, *r5*, *r6*) are multiplied by the temperature factor computed in Equation 2.16, while only *r1* and *r6* are multiplied by the neurotransmitter concentration [38]. The kinetic scheme of the NMDA receptor (Figure 2.15) presents five states, three of which are closed states (C_0, C_1 and C_2), one is the open state O and D is the desensitized state. The general equation for the computation of the receptor current, shown before (Equation 2.20), is slightly modified in the case of the NMDA. In fact, the receptor channel is initially blocked by the magnesium (Mg^{2+}) and this has to be considered in the current computation which is defined as follows (Equation 2.22):

$$I_{NMDA} = \overline{g}_{NMDA} * (V_m - V_{rev,NMDA}) * O(T) * B \tag{2.22}$$

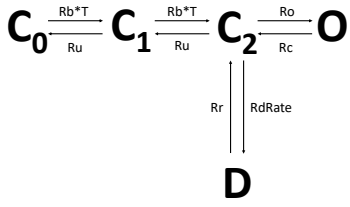


Figure 2.15: NMDA kinetic scheme. 5-state kinetic scheme: C_0, C_1 and C_2, D and O are the *closed*, *open* and *desensitized* state respectively.

where *O(T)* is the open state and B is computed as follows:

$$B = \frac{1}{1 + e^{-\frac{V_m - V_0}{k}}} \tag{2.23}$$

where V_0 is -20 mV and k is 13 mV [38]. The GABA kinetic scheme is more complex than the previous one and it is made up of eight kinetic states, two of which are the open ones (*OA1* and *OA2*). In this case, the current is computed as shown in Equation 2.24, considering the sum of the open states.

$$I_{GABA} = \overline{g}_{GABA} * (V_m - V_{rev,GABA}) * (OA1(T) + OA2(T)) \tag{2.24}$$

Finally, it is possible to conclude that the cellular current can be computed considering both the synaptic (I_{ions}) and the ionic ($I_{synapses}$) currents as follows (Equation 2.25):

$$I = C_m \frac{dV_m}{dt} + I_{ions} + I_{synapses} \tag{2.25}$$

2.6 Golgi and granular cell simulators

At first, the Golgi and granular cells models have been developed by the Neurophysiology laboratory of the University of Pavia exploiting the

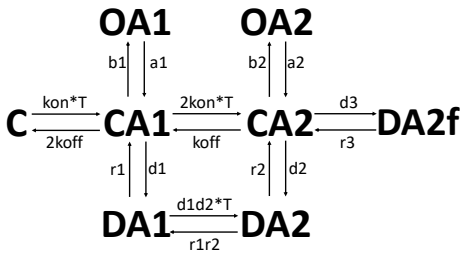


Figure 2.16: GABA kinetic scheme. 8-state kinetic scheme: C, CA1 and CA2 are the *closed* states, DA1, DA2 and DA2f are the *desensitized* states and OA1 and OA2 the *open* states.

Python language since these codes are used in the NEURON simulator. NEURON allows to reproduce the activity of a single cell or of a neuronal network considering also the cell morphology. A negative aspect which characterized this software is the amount of time which takes to run a simulation. As said before, the computational complexity of these realistic models demands a performant technology capable of reproducing the neuron activities with a reduced elaboration time attempting to reach the real-time simulation. As previously explained, this task is very difficult to achieve due to a range of aspects such as the models complexity, the number of neurons that characterizes the simulation (about 100 billion in the human brain [39]) as well as all the hardware characteristics that a technology must be provided in order to perform these heavy simulations (i.e. power supply, computation power, memory size). The first phase of the work involved the development of the algorithm in C language in order to have a serial reference and a first version to exploit parallel technologies easily. This phase does not correspond to a simple algorithm translation from the NEURON files to C language, but requires the development of entire code parts (as the differential equations system resolution), intrinsic in NEURON. Once completed the granular and Golgi cells serial codes, a first parallel version of the models has been implemented exploiting the OpenMP 2.0 API. The aim of these parallel codes is only to evaluate a first parallelization using a multicore approach, exploiting the processor cores in parallel. Therefore, the OpenMP solution exploits simply directives and is not as optimized as the manycore version, written in CUDA C language, which exploits the GPU technology. Even the Field Programmable Gate Array (FPGA) technology has been evaluated in this work. In fact, a circuital model of the granule soma has been designed in order to evaluate the performance of this technology. The circuital model [27], developed with Quartus Prime, has been deployed on an Altera Stratix V FPGA [40]. The simulation carried on this device concerned 3 s of cellular activity of a single granule, considering three different current injections (10, 16, 22 pA as shown in Figure 2.10) and not taking into account the synapses. The simulation satisfied the real-time constraint but the main issue was that all the logic elements available on the board were used only for one cellular

soma. Since the main goal of this work is to simulate the granular layer network, it is clear that a single FPGA is not enough for this workload. For this reason, in the following paragraphs, the parallel versions for the GPU technology of the Golgi and granule cells, also included in the granular layer network, will be presented in detail.

2.6.1 Multi-compartmental Golgi cell simulators

The serial algorithm

As previously said, the serial code was the first algorithm implemented. Its development started from the algorithm designed for the NEURON simulator but has been completely rewritten according to the models presented above, providing the solutions to their differential equations. For each Golgi cell, the algorithm evaluates its synaptic and cellular activity, the current contribution and, at the end, updates the membrane potential. The main flow of the algorithm is shown in Figure 2.17. The *First phase* concerns the parameters initialization. In particular, all the variables related to the Golgi cell are stored in a structure, called *Golgi_cell*. It contains two other structures (one for the excitatory and one for the inhibitory synapse), the membrane potential v, the synaptic current *i_syn*, the ionic channel current and conductance *i_ion* and g respectively, all the gating particles of each ionic channel, and the Nernst potential related to the two different calcium channels *go_eca1* and *go_eca2*. In particular, the two currents are computed as described in the previous paragraphs. The two structures which represent the synapses contain all the parameters related to the presynaptic and postsynaptic terminals models described in Paragraph 2.5.

```
1 struct Golgi_Cell {
2     struct goc_syn_ecc syn_ecc[N_SYN_EXC];
3     struct goc_syn_inib syn_inib[N_SYN_IN];
4     float v; //Membrane potential [mV]
5     float i_syn; //Synaptic current
6     float i_ion; //Ionic channel current
7     float g_ion; // Ionic channel conductance
```

```
 8     float all the gating particles of each ionic channel;
 9     float go_eca1; //Nernst Ca potential (1)
10     float go_eca2; //Nernst Ca potential (2)
11 };
```

Each synaptic structure has an array, called *spike_queue*, which contains all the instants of time in which spikes occur in the presynaptic sites during the simulation. In fact, it is important to highlight that in this phase of the work

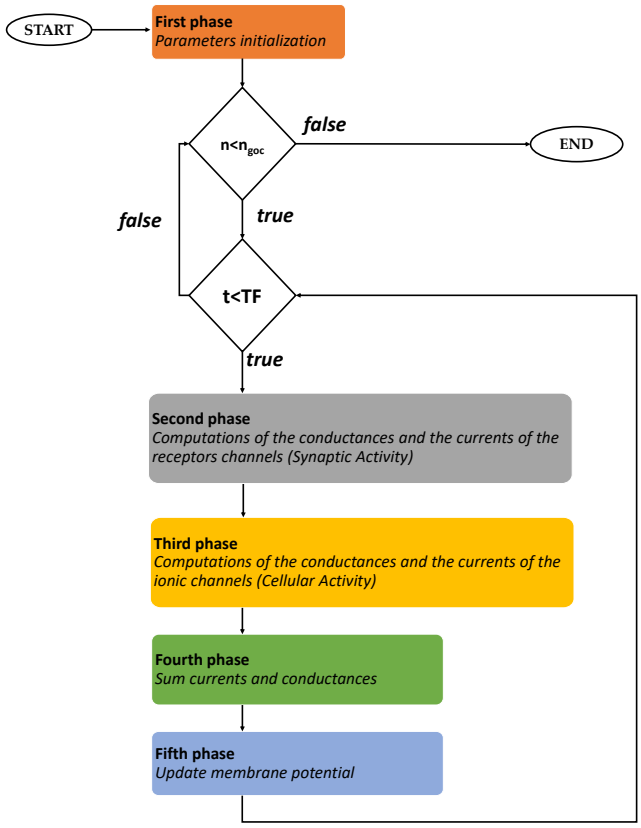


Figure 2.17: Main flow of the Golgi cell algorithm [33].

the cells are independent and they do not communicate. For this reason, the signal exchange is reproduced generating the spikes in the initialization phase through *Spike Train Generator* function. It creates a spikes queue simulating the signals that reach the Golgi cell during its neuronal activity in a network. It is possible to set a range of time in which the function has to generate the action potentials. Furthermore, the frequency with which the spikes occur has to be set: the value is chosen on the base of the typology of the neuron connected to the Golgi cell. The number of spike queues generated for each cell is equal to the number of synapes simulated. Once the synaptic and cell structures are declared, the dynamic allocation of the Golgi structure is performed through a *malloc* function and the amount of memory reserved for this allocation depends on the number of Golgi cells (*n_goc*) that the user wants to simulate. After the *First phase*, the algorithm proceeds with a *for* loop which iterates on the number of cells (*n_goc*) and, then, with the evaluation of the synaptic and cellular activity of the *n*-th Golgi. The pseudo-code of this part is shown in Algorithm 1 [33].

```
1  for n ← 1 to n_goc do
2      for t ← 1 to TF do
3          for e ← 1 to syn_ecc do
4              if spike then
5                  compute glutamate concentration
6                  remove the spike from the queue
7              end
8          end
9          for i ← 1 to syn_inib do
10             if spike then
11                 compute GABA concentration
12                 remove the spike from the queue
13             end
14         end
15         compute INMDA, IAMPA, IGABA and i_syn
16         compute the ionic currents and their sum i_ion
17         check if there are some injected currents in the soma
18         update the membrane potential
19     end
20 end
```

Algorithm 1: Evaluation of the synaptic and cellular activity.

The neuronal activity of each cell lasts *TF* time steps and the discretization step is 0.025 ms according to the value set in the NEURON software. The evaluation of the synaptic activity (*Second phase*) starts computing the amount of neurotransmitter released by the presynaptic terminal that reaches the postsynaptic neuron. In particular, for each excitatory (*lines 3-8*) and inhibitory (*lines 9-14*) synapses, the spike queue is evaluated and, if a spike occurs in the presynaptic terminal, the differential equations system, presented in Paragraph 2.5, is solved and the amount of released neurotransmitter is computed as in Equation 2.18. When this amount is known, the NMDA, AMPA and GABA receptors currents can be computed by solving their kinetic schemes and determining the open state of each scheme (as shown in Paragraph 2.5). Then, they are accumulated in the variable *i_syn* which is in the Golgi structure (*line 15*). This term will be used in the final membrane potential update. The *Third phase* consists in the cellular activity evaluation. The value of the gating particles of each ionic channel is updated and, then, the channel conductances and currents are computed and their values are summed and stored in the variables *g_ion* and *i_ion* (*lines 16-17*). As an example, the code related to the computation of the potassium current and conductance is shown:

```
1 void computeKConductanceCurrent(float v, float e, float *iK,
      float *gK){
2    float an, bn, tau_n, n_inf;
3    ratePotassium(v,&tau_n, &n_inf);
4    an=n_inf/tau_n;
5    bn=1/tau_n;
6    computeN(&n, an, bn);
7    (*gK)= gkbar_Q10*n*n*n*n;
8    (*iK)=(*gK)*(v-e);
9 }
```

At first, the *ratePotassium* function (*line 3*) computes the variables τ_n and n_∞ (in the code *tau_n* and *n_inf*) as shown in Equation 2.11 and Equation 2.12, respectively. Then, *an* and *bn* (*lines 4-5*) are calculated in order to compute the gating particle n value through the function *computeN* (*line 6*), which solves the Equation 2.13. Finally, the conductance *gK* and the current *iK* are computed (*lines 7-8*) as described in Equations 2.9 and 2.6 re-

spectively. Then, the algorithm proceeds checking if some injected currents are present and, in this case, they are considered in the potential update. At the end, the synaptic, the ionic channels and, eventually, the injected currents are considered in the membrane potential update (*Algorithm 1 - line 19*) [33].

The parallel algorithms

As previoulsy said, in this phase of the work, all the Golgi cells are independent. This aspect is crucial in the development of the parallel versions of the Golgi cells algorithm since their neuronal activity can be evaluated at the same time. Concerning the first parallel solution, a multicore strategy has been chosen exploiting the OpenMP API. The CPU threads evaluates the activity of different Golgi simultaneously, reducing the computational time of the simulation. As for the manycore solution, GPUs are the flagship devices for parallel high performance computing on a desktop. In this case, thousands of cores compute in parallel the neuronal activity of the cells. Figure 2.18 shows the Golgi cell algorithm where, in red, the parallelized part is highlighted. The iterations of the first *for* loop, which represent the Golgi cells, are executed in parallel.
Concerning the first OpenMP implementation [33], the idea is that each thread computes the last four phases of the algorithm in Figure 2.18, for a group of cells. Since the number of Golgi cells simulated is *n_goc* and the simulations are carried out on an Intel i7 with eight logical cores, each thread will evaluate the activity of a number of cells equal to *n_goc*/8. The *#pragma* statement allows to generate a parallel region where each thread manages a group of iterations of the *for* loop. In the case of the first OpenMP version (Figure 2.19.A), the loop *#pragma* statement is introduced before the *for* loop which iterates on the cells number:

```
1 #pragma omp parallel for shared(golgi_cell) private(private
      variables) schedule(static)
2 for (j = 0; j < n_goc; j++) {
3     for (t = 0; t < TF; t++) {
4         ...
5     }
```

```
6 }
```

The variable *golgi_cell* is the array of structures presented above and it is defined *shared* so that it is visible by each thread of the parallel region. The variables declared as *private* are the ones referred to the synaptic and cellular activities and that are hidden from the other threads. The *schedule* clause indicates the scheduling policy which will be discussed later. The main difference between this first OpenMP version with the second one is the point where the parallel region is generated. Figure 2.19 shows the comparison between these two codes. In the first case, the parallel region

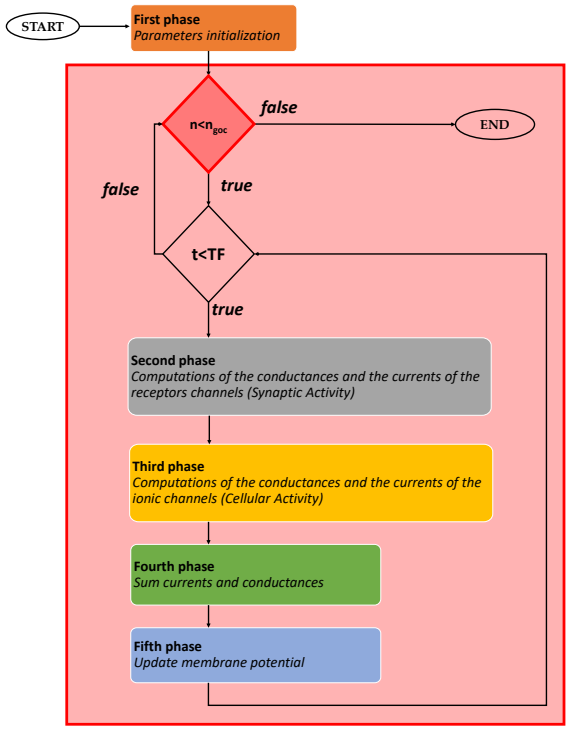


Figure 2.18: Main flow of the Golgi parallel algorithm. The figure shows the main steps of the Golgi algorithm. In the red box there is the part of the code that is parallelized.

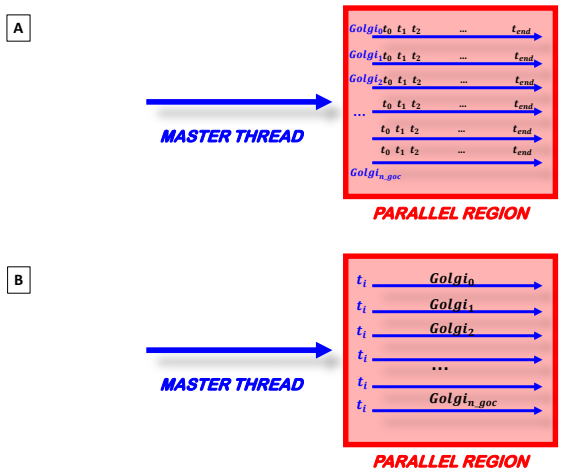


Figure 2.19: Comparison between two OpenMP versions. A) First OpenMP version [33]; B) Second OpenMP version.

has been generated before the *for* loop related to the cells (Figure 2.19.A): for this reason, as previously said, each thread computes the activity of a group of Golgi cells. In the second case (Figure 2.19.B), the two loops are swapped and the parallel region is generated after the *for* loop related to the simulation time. This means that each thread computes a single discretization step (t_i) of the activity for all the cells. In this case the two *for* loops are developed as follows:

```
1 for (t = 0; t < TF; t++) {
2 #pragma omp parallel for shared(golgi_cell) private(private
      variables) schedule(static)
3     for (j = 0; j < n_goc; j++) {
4         ...
5     }
6 }
```

At the end, a variant of this last version has been developed in order to reduce the parallelization overhead. In this case the *for* loop related to the time of the simulation has been replaced with a *while* loop. The parallel

region is generated before the *while* loop, as shown in the following code:

```
1 #pragma omp parallel for shared(golgi_cell) private(private
      variables)
2 while (t < TF) {
3   #pragma omp for schedule(static)
4   for (j = 0; j < n_goc; j++) {
5       ...
6   }
7 }
```

The main difference is that in the second OpenMP version the parallel region is generated during each iteration of the *for* loop. In the third version, this region is created once before the *while* loop. In the following paragraph, the results of these three multicore versions will be discussed, evaluating different scheduling types (*static*, *dynamic*, *guided* and *runtime*).
To exploit the manycore phylosophy and the GPU technology, a Golgi algorithm version written in CUDA C language has been developed. Figure 2.20 shows the main flow of the parallel algorithm. The flow starts on the host where the parameters initialization of the *First phase* is performed. Unlike the serial code, this version includes a further step which concerns the data packing in arrays that have to be transferred on the device. This phase is crucial to reach high performance and to reduce the computational times. In fact, if not properly managed, the data transfer could be the bottleneck of the process. In order to prevent this potential slow-down, all the data related to the cells have been stored at contiguous memory addresses trying to minimize the bus activations during the transfer. The idea is to create a 1D array and to join the data according to their physiological meaning. For example, considering the spike queues presented above, the adopted strategy in the data packaging is shown in Figure 2.21.

It is important to consider that, in this case, the word *synapse* is not referred to a single connection between two neurons. Here it indicates the totality of the connections that a Golgi cell makes on the basis of where the connections are located. In fact, the Golgi cells present two excitatory synapses that represent the connections that they make in their apical and

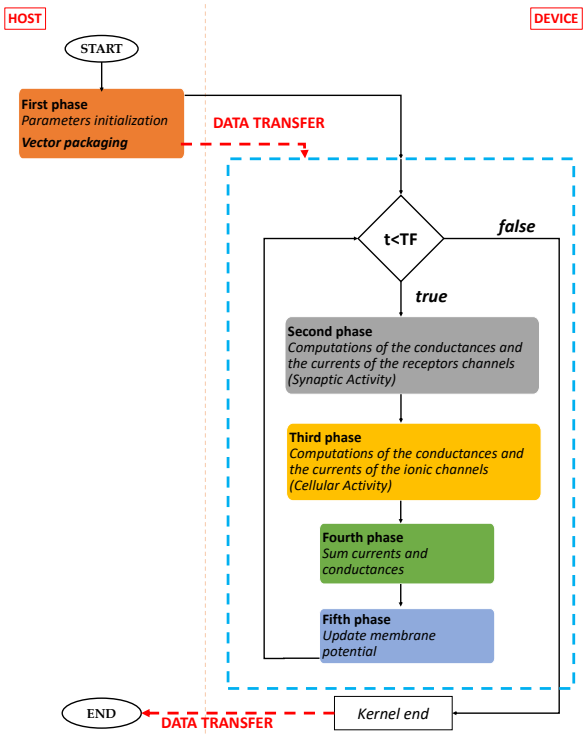


Figure 2.20: CUDA parallel algorithm [33].

basal dendrites and a synapse which represents the inhibitory connections. So, it is possible to say that in this model the Golgi cell is characterized by three synapses (two excitatory and one inhibitory) while always keeping in mind that the element *synapse* includes hundreds of connections that a Golgi cell performs in the reality.
As said before, in the model each synapse is characterized by a *spike queue*. In order to transfer all the spikes related to all the cells synapses, the strategy is to create an array which contains all these data organized so as for each cell, the spikes related to the excitatory synapses (in green in Figure 2.21) are firstly stored, followed by the ones related to the inhibitory one (in red) [33]. This strategy, illustrated for the spike queues, has been

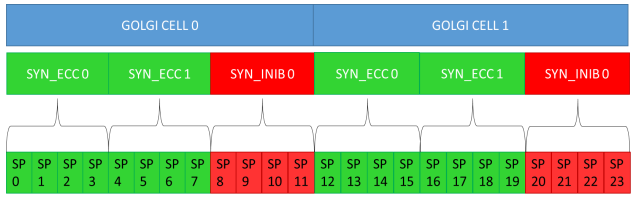


Figure 2.21: Data packing. All the data are stored at contiguous memory addresses to improve the performance. This figure shows the case of the spike queues packaging [33].

followed for all the data transfers from host to device. Once the data have been transferred and stored in the device global memory, they have to be copied in the shared memory so that they can be quickly accessed by all the threads in the block. The block dimension is set as a multiple of 32, according to the *warp* dimension, in order to optimize the scheduling carried out by the NVIDIA Giga Thread scheduler [33]. For this reason, each block contains all the data related to the Golgi cells whose activity is evaluated by the threads in that block.
After storing all the data in the blocks local memory, the computation can start. A kernel (represented in the light blue dotted box in Figure 2.20) is generated with a grid dimension defined according to Equation 2.26.

$$\dim_{grid} = \frac{n\_goc}{n_{threads}} \tag{2.26}$$

where n_goc is the number of Golgi cells and $n_{threads}$ is the number of threads that are in one block of the grid. In this case, this number is set to 32. If the remainder of Equation 2.26 is not equal to zero, dim_{grid} is incremented by one. Since the kernel evaluates simultanously the activity of different cells, the first *for* loop presented in the previous versions is not necessary. Each thread of the kernel evaluates the complete neuronal activity of the Golgi cell, starting from the *Second phase* which concerns

the synaptic activity described above. Once the receptors currents have been computed, the cellular activity evaluation can start (*Third phase*). As described above, all the gating particles values are updated in order to compute the ionic channels conductances and currents. Then, the algorithm evaluates if some injected current stimulates the cell. If so, this term will be considered with the other currents in the *Fifth phase*, where the membrane potential is updated. For each cell, all the updated membrane potential are stored in the global memory at contiguos addresses in order to follow the previously explained strategy. In fact, these values are then transferred from the device to the host where the final results are written.
A second CUDA parallel version has been developed in order to both exploit a multi-GPU system and to analyse the scalability on multiple devices. The idea is to divide the evaluation of the neuronal activity of the cells into two groups, each one performed by a GPU, as shown in Figure 2.22. In this way, each device has to evaluate the neuronal activity of $n\_goc/2$ cells. After the parameters initialization, the data packing is performed by storing the data into two 1D arrays, one for each GPU. In Figure 2.22, the data transferred from host to device 0 are indicated with a green dotted arrow, while those sent to device 1 with the red one. The data transfers on the two devices are performed calling the function *cudaMemcpyAsync* using *streams*, which allow to call the GPU functions in an asynchronous and non-blocking way, so that the two devices can work in parallel. It is important to underline that, even if a multi-GPU system is used, in this case the GPUDirect tool is not needed since the two devices do not have to communicate. At the beginning, the workload is separated and the computation performed on the device 0 is independent from the one managed by the device 1. After the data transfers, the kernels are executed one on each GPU, always using streams. The phases contained in each kernel are the same described above for the single-GPU code. Once the neuronal evaluation of all the cells finishes, the results are transferred from the devices to the host.

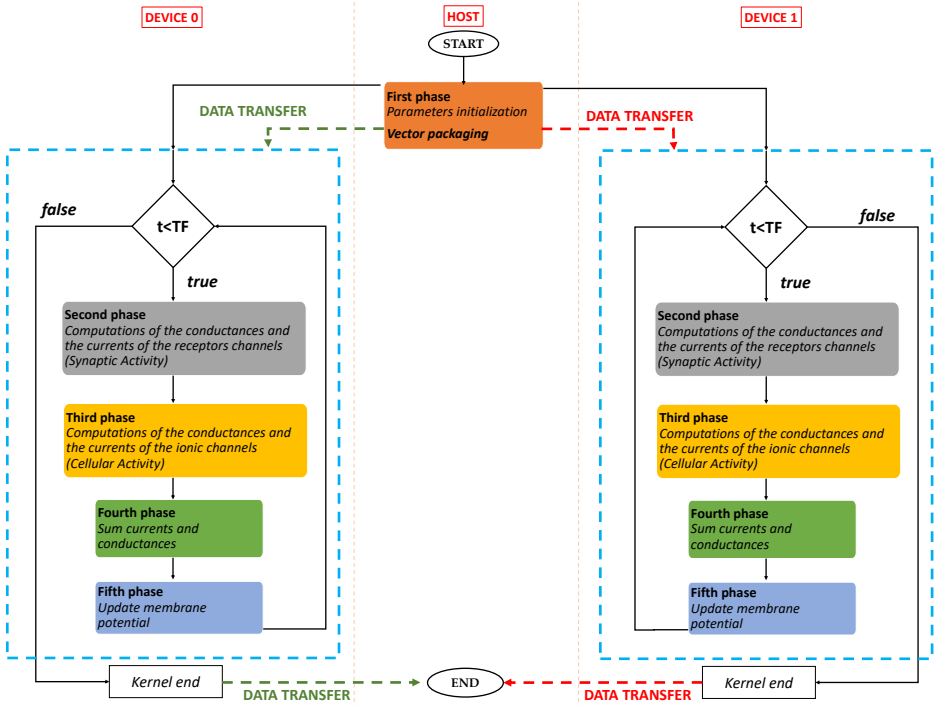


Figure 2.22: Golgi algorithm for multi-GPU system.

Results

The Golgi cell multicores and manycores codes have been tested on *System 1* and *System 2* presented above. In particular, the multi-GPU code is also tested using both the GPUs of *System 1*. All the codes have been compiled using Microsoft Visual Studio 2015 environment and the *nv140 compiler*. Concerning CUDA GPU version, the environment used is the 8.0. Microsoft Visual Studio C compiler supports the OpenMP 2.0 paradigms. OpenMP support has been enabled using the flag */openmp*. The computational times presented in this paragraph are the average of five executions and they refer to simulations of 3 s of neuronal activity either in vivo (from 0 to 1,500 ms) and in vitro (from 1,500 to 3,000 ms). This

Cell number	Serial - System 1 [s]	Serial - System 2 [s]
1	0.22	0.20
5	0.87	0.70
10	1.68	1.36
50	8.25	6.58
100	16.23	13.12
500	81.29	65.95
1,000	163.21	130.33
5,000	811.13	648.67
10,000	1,636.16	1,306.96
25,000	4,051.93	3,251.67
50,000	8,101.58	6,644.69
100,000	16,321.40	13,035.67
200,000	32,863.56	30,086.51
400,000	66,406.62	51,793.64

Table 2.1: Computational times of the serial algorithm. These computational times concerns the serial algorithm executed on the *System 1* (equipped with a Intel i7 3770, 4 cores, 8 GB RAM and 3.40 GHz of working frequency) and on the *System 2* (equipped with a Intel i7 6700, 4 cores, 32 GB RAM and 3.40 GHz of working frequency).

means that in the first half of the simulation the synaptic activity is present while in the second part there are only current injections without synaptic activity. The computational times of the serial algorithm are shown in Table 2.1.

As it is possible to see in Table 2.1, the evaluation of the activity of 400,000 cells takes respectively about 18 and 14 hours if *System 1* or *System 2* are considered. This result confirms that the simulation of the neuronal activity of a huge amount of cells, described by a very complex models, takes a lot of time. The final simulation of the granular layer network will contain a higher number of cells and, for this reason, it is crucial to exploit HPC technologies in order to increase the performances. Another analysis that can be done is that the processor of *System 2* is more efficient than *System 1* one. The reason is the different architecture of the two processors. In particular, *System 1* is equipped with a processor based on Intel 3rd generation Core architecture (*Ivy Bridge*) while *System 2* features

a device based on Intel 6th generation Core architecture (*Skylake*).

Table 2.2 and Table 2.3 show the computational times of the three OpenMP algorithms exploiting *System 1* and *System 2*. Both the systems are characterized by a processor with 4 physical and 8 logic cores. Each code presents four versions where the scheduling type changes (*static*, *dynamic*, *guided* and *runtime*). As described before, the first OpenMP version performs in parallel the neuronal activity evaluation, generating a parallel section before the *for* loop which iterates on the cells number. The second version computes each discretization step of the activity simultaneously for all the cells. The third version follows the same strategy with the exception of the parallel region that is not generated at each iteration but only once. What is important to notice in these tables is that the multicore strategy provides a speed-up when compared to the serial version. In particular, if the three algorithms are considered, the OpenMP - Version 1 with the *guided* scheduling achieves the best results both considering *System 1* and *System 2*. This version introduces a lower parallelization overhead than the other two: in fact, in OpenMP - Version 2 the parallel region is generated during each iteration, while the OpenMP - Version 3 presents nested *pragmas* and the Microsoft compiler is less efficient in the code generation. Table 2.4 shows the computational times related to the single-GPU CUDA version performed on *System 1* (equipped with the Tesla K40) and on *System 2* (equipped with the GTX1060). Moreover, the table presents the multi-GPU results considering the simulation carried on the two Tesla K40 board contained in *System 1* but also on one node of the CINECA supercomputer named Galileo [41]. Each node of Galileo is equipped with two GPU NVIDIA K80 [42] characterized by 4992 CUDA cores with a dual-GPU design, 24 GB of GDDR5 memory and 480 GB/s memory bandwidth.

Analyzing results in Table 2.4 it is clear that the manycore solution is the best performing since it significantly reduces the computational times of the serial and multicore solutions. The simulation of the neuronal activity of 400,000 cells on *System 1* takes about 28 minutes instead of about 18 hours of the serial code and about 4 hours of OpenMP - Version 1. It is also important to observe that *System 2* provides good results in the

	OpenMP - Version 1 [s]				OpenMP - Version 2 [s]				OpenMP - Version 3 [s]			
Cell number	**Static**	**Dynamic**	**Guided**	**Runtime**	**Static**	**Dynamic**	**Guided**	**Runtime**	**Static**	**Dynamic**	**Guided**	**Runtime**
1	0.24	0.22	0.25	0.23	0.42	0.44	0.45	0.42	0.32	0.35	0.37	0.43
5	0.31	0.28	0.27	0.31	0.46	0.51	0.51	0.57	0.44	0.48	0.46	0.46
10	0.52	0.46	0.63	0.56	0.76	0.70	0.74	0.78	0.69	0.67	0.74	0.75
50	1.85	1.87	1.88	1.83	2.14	2.18	2.14	1.94	2.11	2.14	2.21	2.27
100	3.57	3.60	3.65	3.60	3.67	4.09	3.99	4.14	3.94	3.99	4.10	3.51
500	17.32	17.57	17.25	17.48	18.10	18.36	18.65	18.63	18.36	18.33	18.35	15.89
1,000	33.31	35.08	34.42	34.29	35.47	36.16	35.20	35.71	35.78	36.06	40.01	29.22
5,000	176.12	174.58	173.91	175.68	172.43	178.06	180.60	180.95	186.94	179.68	187.53	189.57
10,000	344.33	336.83	342.68	339.24	390.75	379.44	372.08	377.77	395.99	402.69	371.70	392.80
25,000	837.88	871.37	829.35	834.28	1,051.96	1,047.58	1,044.18	1,048.26	1,067.91	1,073.83	1,061.54	1,068.37
50,000	1,716.64	1,679.91	1,718.47	1,745.88	2,151.52	2,103.66	2,101.24	2,106.24	2,172.65	2,184.56	2,179.69	2,189.29
100,000	3,307.06	3,356.47	3,301.06	3,315.44	4,343.74	4,316.61	4,312.55	4,315.24	4,362.06	4,371.34	4,360.29	4,369.71
200,000	6,807.04	6,909.13	6,799.95	6,814.14	8,670.52	8,740.24	8,674.23	8,671.71	7,790.54	8,439.22	7,781.71	7,794.36
400,000	13,749.15	13,966.77	13,720.18	13,759.59	16,185.43	17,241.13	16,184.23	16,189.57	17,458.43	16,128.26	17,446.29	17,459.28

Table 2.2: Computational times of the OpenMP versions obtained with *System 1*.

		OpenMP - Version 1 [s]				OpenMP - Version 2 [s]				OpenMP - Version 3 [s]		
Cell number	**Static**	**Dynamic**	**Guided**	**Runtime**	**Static**	**Dynamic**	**Guided**	**Runtime**	**Static**	**Dynamic**	**Guided**	**Runtime**
1	0.20	0.23	0.23	0.21	0.29	0.31	0.36	0.31	0.28	0.32	0.34	0.34
5	0.24	0.24	0.33	0.30	0.40	1.19	1.24	0.42	0.36	1.14	1.18	0.41
10	0.47	0.44	0.49	0.51	0.66	1.26	1.35	0.70	0.60	1.24	1.30	0.66
50	1.80	1.76	1.75	1.84	1.98	2.93	2.66	2.01	2.08	2.98	2.60	2.09
100	3.10	3.33	3.38	3.14	3.90	5.03	4.25	3.95	3.95	4.84	4.37	3.68
500	16.27	16.70	15.99	16.35	17.58	20.16	16.69	17.70	18.02	19.99	17.42	17.28
1,000	29.66	32.73	33.42	30.01	33.52	38.07	33.47	33.80	35.46	40.28	33.36	35.09
5,000	163.82	163.57	156.81	163.95	172.36	195.53	163.02	177.24	176.85	200.08	166.92	182.21
10,000	323.86	321.95	327.87	324.34	358.08	430.52	335.62	374.23	371.77	403.51	374.25	369.52
25,000	755.20	812.90	756.14	758.29	891.26	1,068.78	921.07	901.54	929.94	1,085.74	909.83	926.80
50,000	1,667.93	1,645.83	1,631.23	1,674.25	1,903.29	2,036.63	1,837.21	1,908.03	1,965.76	2,131.26	1,876.85	1,968.27
100,000	3,280.58	3,189.86	3,170.54	3,284.03	3,666.82	4,041.02	3,493.18	3,670.37	3,748.30	4,282.17	3,719.17	3,750.15
200,000	6,498.70	6,476.11	6,458.47	6,480.54	7,492.25	8,385.7	7,450.55	7,506.14	7,592.10	8,191.52	7,540.37	7,608.59
400,000	12,886.06	12,953.47	12,885.74	12,890.45	14,757.79	16,527.99	14,977.07	14,780.68	15,453.64	17,094.64	14,287.68	15,498.73

Table 2.3: Computational times of the OpenMP versions obtained with *System 2*.

	Single GPU		Multi GPU	
Cell number	**CUDA - System 1 [s]**	**CUDA - System 2 [s]**	**CUDA - System 1 [s]**	**CUDA - Galileo (CINECA) [s]**
1	8.36	5.24	-	-
5	8.45	5.24	8.39	-
10	8.54	5.27	8.41	-
50	8.88	5.28	8.84	-
100	8.86	5.31	8.81	7.57
500	8.32	5.62	8.72	7.18
1,000	8.08	5.83	8.16	7.14
5,000	24.40	24.68	15.69	6.81
10,000	47.97	88.26	24.28	13.12
25,000	108.94	107.53	54.91	19.89
50,000	209.46	-	108.67	45.61
100,000	411.14	-	209.25	83.29
200,000	816.23	-	410.73	159.63
400,000	1,675.70	-	898.55	313.10

Table 2.4: Computational times of the single-GPU CUDA algorithm.

	System 1			***Cineca***	***System 2***	
Cell number	**OpenMP Version 1 [s]**	**CUDA Single GPU [s]**	**CUDA Multi GPU [s]**	**CUDA Galileo [s]**	**OpenMP Version 1 [s]**	**CUDA Single GPU [s]**
1	0.88	0.03	-	-	0.86	0.04
5	3.22	0.10	0.10	-	2.12	0.13
10	2.67	0.19	0.20	-	2.77	0.26
50	4.38	0.93	0.93	-	3.76	1.25
100	4.45	1.84	1.84	2.14	3.88	2.47
500	4.71	9.77	9.32	11.32	4.12	11.73
1,000	4.74	20.20	20.00	22.86	3.89	22.35
5,000	4.66	33.24	51.70	119.11	4.13	26.28
10,000	4.77	34.11	67.39	124.70	3.98	14.81
25,000	4.88	37.19	73.79	203.72	4.30	30.24
50,000	4.71	38.68	74.55	177.63	4.07	-
100,000	4.94	39.70	78.00	195.96	4.11	-
200,000	4.83	40.23	79.95	205.87	4.65	-
400,000	4.84	39.63	73.90	212.09	4.02	-

Table 2.5: Speed-up *System 1*, *Galileo (Cineca)* and *System 2*

execution of the CUDA code but it is not able to simulate the activity of a number of cells higher than 25,000. This is due to the limited amount of memory of this GTX 1060 board. The performances further grow up if the multi-GPU algorithm is considered. The simulation of 400,000 cells takes about 15 minutes in case of *System 1* and only 313.10 s considering the Galileo supercomputer. As it is possible to see from the table, tests with low number of cells are not executed since the multi-GPU systems increase their performance with a great amount of neurons to simulate. Finally, to facilitate the comparison of the results, Table 2.5 shows the speed-up of OpenMP - Version 1 (with *guided* scheduling), of the single GPU and, for *System 1*, of the multi-GPU algorithms with respect to the serial code.

As previously said, the multicore solution provides reduced computational times compared to the serial version, both considering *System 1* and *System 2*. This first parallelization allows to reach a speed-up equal to 4.94×. If the manycore solution is considered, the computational times are further reduced, obtaining speed-up that reach 40.23× for the single-GPU algorithm and 79.95× for the multi-GPU (*System 1*). Moreover, if the simulation performed on the supercomputer Galileo is considered, the speed-up reaches 212.09×. Analyzing these results, it is clear that to achieve a real-time simulation, considering these complex mathematical models, a huge computing power is needed. In this context, the multi-GPU systems are an efficient technology to significantly reduce the elaboration times.
At the end, it is crucial to observe that the serial and the developed parallel algorithms can correctly reproduce the Golgi cell behaviors presented above. In particular, Figure 2.23.A shows the natural pacemaking that the Golgi cell presents at rest. If the cell is stimulated with two current injections, it increases the frequency in the spikes generation: Figure 2.23.B shows the response to a 100 pA current injected from 1,000 to 2,000 ms and to a 200 pA current injected from 2,000 to 3,000 ms. As said before, as the current amplitude increases, also the frequency of the spikes occurrency grows up. These algorithms are even able to reproduce the *rebound excitation* (Figure 2.23.C) and the *phase reset* (Figure 2.23.D) described above. In particular, the *phase reset* has been simulated considering six cells firing at different phases: after a specific injection the cell starts to spike simultaneously.

2.6.2 Mono-compartmental granular cell simulators

[1] The serial and parallel versions of the granule have been previously implemented by the author of this PhD work during her master thesis. In this paragraph a short explanation of this algorithm is given since it will

[1]The contents of this paragraph are published in *Florimbi G., Torti E., Masoli S., D'Angelo E., Danese G. and Leporati F., "The human brain project: parallel technologies for biologically accurate simulation of granule cells"*, https://doi.org/10.1016/j.micpro.2016.05.015, Creative Commons CC-BY-NC- ND license, Microprocessors and Microsystems, 2016.

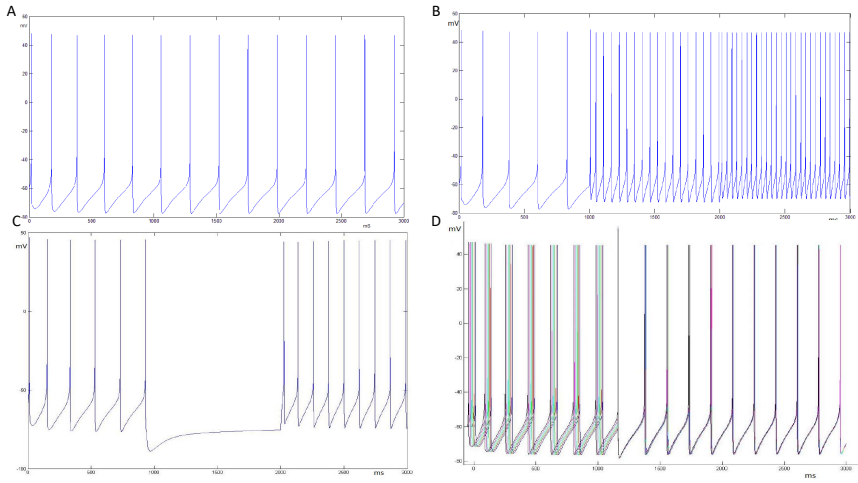


Figure 2.23: Golgi cells behaviors. These graphs have been generated with the developed simulator. A) Natural pacemaking of the Golgi cell; B) Golgi cell response to two different current injections; C) rebound excitation; D) phase reset.

be included in the granular layer network, presented below.

The serial algorithm

The mainflow of the serial algorithm which reproduces the granule cell model is shown in Figure 2.24. In the algorithm implementation, the same strategy used for the Golgi cell has been used. In fact, it is possible to notice that the phases that characterized the algorithm are the same. The main differences are the typology of ionic channels present in the cellular soma and the number of synapses. The *First phase* concerns the parameters initialization and sets the variables which describe the synapses and the granules behaviour. Furthermore, in this phase, conductances are multiplied by a corrective factor which represents temperature effects. The flow proceeds with a loop which describes each granule cellular activity. This iteration continues until $t < TF$ where t is simulation time and TF is the last instant of the cellular activity. The first phase of this cycle is made up of

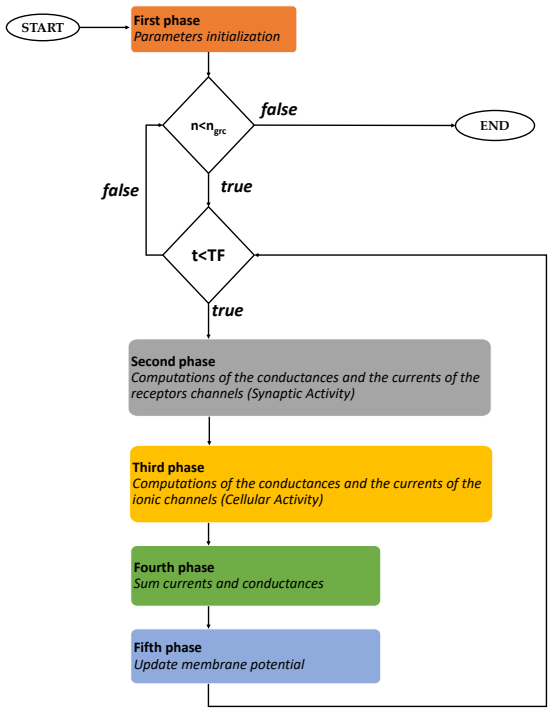


Figure 2.24: Main flow of the granular cell algorithm.

the following steps, that compute:

1. the glutamate amount released by the presynaptic terminal;
2. kinetics states of the AMPA and NMDA channels;
3. conductances and currents of these receptors;
4. the GABA amount released by the presynaptic terminal;
5. kinetics states of the GABA channels;
6. conductances and currents of this receptor.

Finally, all the excitatory and inhibitory synaptic currents are computed as described in Equations 2.20, 2.22 and 2.24 respectively. The successive step (*Third phase*) is characterized by the computation of the conductances and currents of the ionic channels, presented in Paragraph 2.4.1, described by the gating particles model. All the ionic and leakage currents and conductances are summed and stored in suitable variables. The *Fourth phase* consists in the sum of the synaptic and ionic conductances and currents computed at the previous steps. At the end (*Fifth phase*), the membrane potential is updated as in Equation 2.25. This algorithm allows to perform two different simulations: the first one, called in vitro, shows how the neuron reacts to a current injection in the soma, without taking into account synaptic activity. The variable which describes the injected current amplitude is added to the other currents. In this kind of simulation the synaptic current is of course forced to 0 pA (no synaptic activity in *Second phase*). On the other hand, an in vivo simulation should be performed, to understand how the neuron reacts to spikes generated by the presynaptic terminal while there is no current injection.

The parallel algorithms

The loop iterations present in Figure 2.24 and describing the cellular activity of each granule are independent. For this reason, it is possible to execute these iterations simultaneously using the multicore and manycore strategies described above. Here again, as for the Golgi parallel algorithms, the part of the flow that can be parallelized is the one inside the first *for* loop which iterates on the number of cells. Concerning the multicore strategy, the OpenMP 2.0 API has been exploited. Also for the granule code, each thread calculates the cell activity of a granules group. The algorithm has been modified introducing suitable *#pragma* statements and specifying which variables are private or shared regards to the other threads. This implementation is tested on an Intel i7 processor with four physical cores (eight logical cores), therefore each thread computes the cell activity of $N\_CEL/8$ granules, where N_CEL is the number of simulated granules. As previously said, to achieve further acceleration, the algorithm has been

developed in CUDA C exploiting the GPU technology. Also in this case the idea is to execute the loop iterations in parallel: each thread calculates the activity of each granule simultaneously. Also for the CUDA algorithm development, the strategy adopted is the same used in the Golgi code. The GPU parallel execution can be resumed in these three main phases:

1. device memory allocation and variable transfer from host to device;
2. kernel execution;
3. results transfer from device to host and device memory deallocation.

In particular, as the GPU is initialized and once the data are transferred from host to device, a unique kernel is launched. The kernel contains all the phases inside the time loop presented in Figure 2.24. In fact, each thread evaluates the neuronal activity of a single granule cell. At the end of the computation, the membrane potential values are transferred from device to host. The memory transfers from host to device and viceversa are time consuming processes that have to be properly managed. All the transferred data are stored in 1D array consecutive locations allowing a considerable gain of time.
Also in the case of the granular cell model, a multi-GPU algorithm has been developed following the same strategy described for the Golgi cells. The workload has been divided among two devices. For this reason, at the beginning of the computation, the host transfers data both to device 0 and device 1 and the flow proceeds as shown in Figure 2.22 for the Golgi algorithm.

Results

The serial, OpenMP and CUDA versions have been executed on *System 1*, already presented in Paragraph 1.2.2. Also in this case the multi-GPU algorithm has been tested also on the Galileo supercomputer. The simulated cell activity lasts 3 s: in the first half there is a current injection (without synaptic activity) while in the remaining one there is only synaptic activity. Simulations have been conducted with an increasing granule

# Granules	Serial [s]	OpenMP [s]	*System 1* Single GPU [s]	*System 1* Multi GPU [s]	Galileo Multi GPU [s]
1	0.34	0.36	10.11	-	-
5	1.71	1.00	10.15	-	-
10	3.34	2.06	10.17	-	-
50	16.70	8.97	18.75	-	-
100	33.54	16.54	18.77	-	-
500	167.11	83.95	36.85	-	-
1,000	334.15	175.99	55.55	31.79	14.74
5,000	1,646.17	837.12	205.04	94.55	14.78
10,000	3,243.43	1,684.89	393.76	177.43	28.79
25,000	8,813.9	4,266.81	975.40	429.37	71.92
50,000	16,225.42	7,902.71	1,949.47	846.56	153.09
100,000	32,620.71	16,505.73	3,892.24	1665.51	288.95
200,000	65,282.04	30,524.79	7,778.74	3318.40	564.44
400,000	129,894.74	67,268.86	15,580.25	6645.82	1126.59

Table 2.6: Granular cells serial and parallel algorithms results. The table presents the computational times obained with the execution of the serial, OpenMP and CUDA codes. The times are expressed in seconds and refers to a cellular activity evaluation of 3 s.

number, from 1 to 400,000, and an increasing synapse number, from 8 to 3,200,000 (8 synapses for each granule). Serial and OpenMP execution times are lower than the CUDA ones for simulations of a few granules. If the simulated cells number grows up, GPU is the best solution as it can be seen in Table 2.6, where the multi-GPU computational times are also presented. It is important to highlight that the evaluation of 400,000 granules activity takes about 36 hours if the serial code is considered. This time is considerably reduced if the CUDA code for a single GPU is used: it takes about 8 hours. A further reduction of the elaboration times is obtained with the multi-GPU systems: in fact, *System 1* takes less than 2 hours and the Cineca supercomputer takes about 18 minutes. Notice that *System 1* is equipped with a total of 5760 cores, while one node in the Galileo systems contains a total amount of 11,520 cores. This computational power allows to reach the speed-up shown in Table 2.7. These values prove that the GPU is an efficient technology to perform this kind of simulation, providing great speed-up especially if the multi-GPU systems are considered. For example, as for 400,000 neurons, the speed-up reaches 19.67× and 115.65× if *System 1* and Galileo are used. As the number of simulated cells increases the

# Granules	OpenMP [s]	*System 1* Single GPU [s]	*System 1* Multi GPU [s]	Galileo Multi GPU [s]
1	0.93	0.03	-	-
5	2.00	0.17	-	-
10	1.62	0.33	-	-
50	1.86	0.89	-	-
100	2.03	1.79	-	-
500	1.99	4.53	-	-
1,000	1.89	6.01	10.51	11.38
5,000	1.97	8.03	17.40	22.67
10,000	1.92	8.24	18.28	111.38
25,000	1.91	8.39	19.06	112.66
50,000	2.05	8.32	19.16	113.79
100,000	1.98	8.38	19.58	105.99
200,000	2.14	8.39	19.67	112.89
400,000	1.93	8.34	19.54	115.65

Table 2.7: Granular cells serial and parallel algorithms speed-up. The table presents the speed-up obained comparing the OpenMP and CUDA codes with the serial algorithm.

speed-up calculated between the serial code and the GPU versions grows up until it becomes constant because sequential code parts prevail on the parallel ones. Higher GPU performances are due to massive parallelism of the algorithm and also to the optimization of variables transfer described before.

This type of simulation is very useful for two main reasons: to have a parallel version of the algorithm, which will be included in the network and to evaluate in parallel the neuronal activity of a huge amount of cells that could be stimulated by different physiological protocols (injected currents profiles). Finally, Figure 2.25 shows that also these algorithms are able to correctly reproduce the cellular response to three different current injections. The first current (10 pA) is under threshold and, for this reason, the granule does not generate spikes. The other two currents (16 and 22 pA) excite the cell that shows the action potentials with an increased frequency due to the increased current amplitude.

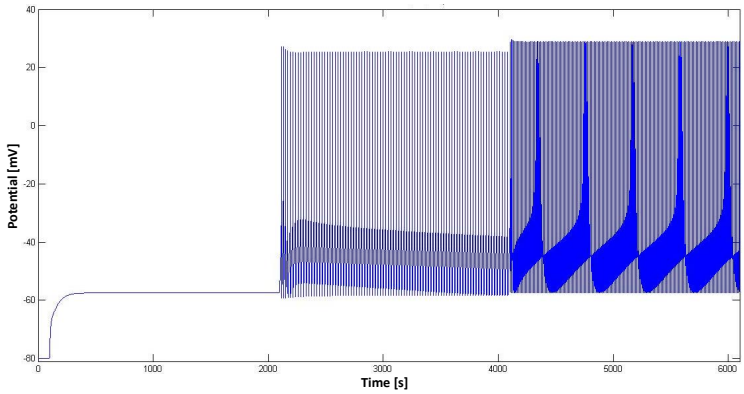


Figure 2.25: Granule cell simulation results. The cell is stimulated by three different current injection (10, 16, 22 pA). In this case a cellular activity evaluation of 6 seconds has been performed to compare the results with the reference graph shown in Figure 2.10.

2.7 The cerebellar granular layer network

In the first part of the work, presented in the previous paragraphs, the granules and Golgi cells simulators have been developed. This phase was crucial to evaluate an efficient technology capable of significantly reducing the elaboration times of the serial algorithms. Moreover, the parallel algorithms developed in this step will be used in the granular layer network described in this paragraph. As presented in Paragraph 2.2, the cerebellar granular layer hosts the granule and Golgi cells which connect their dendrites and axons in structures called glomeruli, reached also by the mossy fibers [35]. The aim of this phase of the work is to create a granular layer reproduction, considering the connection rules to realistically link neurons, including the granule and Golgi simulators described above. The development of this neuronal network consists in two main phases:

- network design;
- neuronal network actvity simulation.

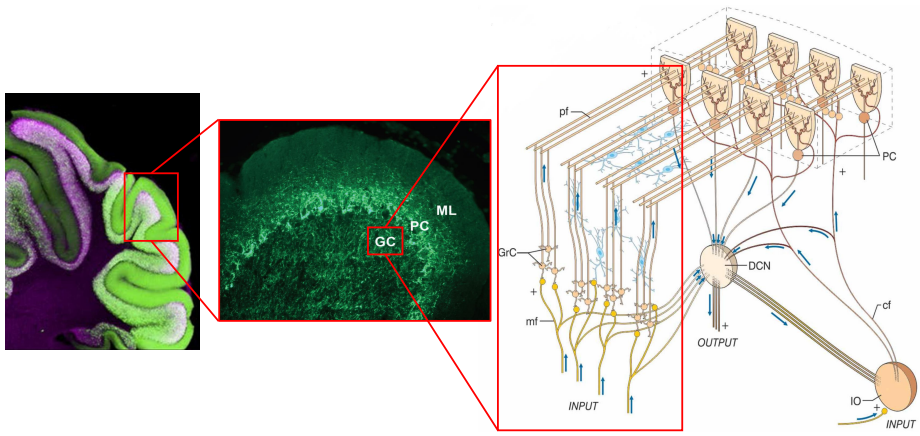


Figure 2.26: Cerebellar neuronal network. The Figure on the left shown the cerebellar cortex, that is made up of three layers: the granular (GC), the Purkinje (PC) and the molecular (ML) layer, represented in the central image. On the right it is possible to see the model which riproduces these three layers.

The *network design* phase consists in placing different type of neurons as realistically as possible in a 3D space, taking into account their cellular morphology. Moreover, in this step the neurons connection has been implemented following defined rules which try to reproduce the biological interactions that occur between different cells. Once the elements are placed and connected, their neuronal activity is simulated exploting the cellular simulators (*neuronal network actvity simulation* phase).
Also in this case, at first the serial algorithm of the network design and of the activity simulation has been developed in C language. Then, a parallel version of the simulation part has been implemented to exploit the GPU technology.

2.7.1 Network design

The granular layer network structure has been generated following detailed anatomical and functional information. It is important to highlight that the basic idea followed in the network development has been to con-

	Dimension [μm]
LENGHT	600
HEIGHT	150
DEPTH	1200

Table 2.8: Volume size.

struct a *non-fixed structure*. This means that even though the network is defined by specific structural and connection rules it is still possible to modify its size and to change the dimension of several physiological elements (such as soma, dendrites, axons). In fact, the aim is to build a *parametric* network which can reproduce different configurations only by changing the variables values. The volume that will be reproduced in this work is characterized by the dimensions presented in Table 2.8. As previously said, the user can change these values and test new configurations. To the best of the author knowledge, this network configuration includes anatomical details and parameters values that are the most relevant in the literature. The advantage of considering a non-fixed structure is also crucial in order to change the network configuration when new anatomical details or values will be discovered. On the other hand, this flexibility should be intended only in terms of parameters variability rather than inserting new constraints.
The *network design* phase and, therefore, its serial algorithm, consists of three main steps:

- compute the elements number;
- displace the elements in a 3D space;
- connect the elements.

First of all, the number of granules, Golgi cells and glomeruli has to be computed considering their density in a volume. In this particular simulation, the used density values are the ones indicated in [35] and referred to a rat but, as said before, the user could change these values if new data will be provided. Table 2.9 shows the density values and the number of elements that this volume can host. The number of cells has been obtained by

dividing the elements density by the considered volume. Once the elements number has been computed, the flow proceeds with the dynamic allocation of the arrays which contain the Golgi cells, granules and glomeruli coordinates and all the connection schemes, that will be discussed later. The arrays *c_glo*, *c_goc* and *c_grc* contains the elements coordinates and they are allocated through the *malloc* function as follows:

```
1  c_glo=(float *)malloc((N_COORD*n_glo)*sizeof(float));
2  c_grc=(float *)malloc((N_COORD*n_grc)*sizeof(float));
3  c_goc=(float *)malloc((N_COORD*n_goc)*sizeof(float));
```

where *N_COORD* is the number of coordinates and *n_glo*, *n_goc* and *n_grc* are the number of glomeruli, Golgi cells and granules respectively. The first version of the elements displacement algorithm placed the neurons and the glomeruli in a *random* way while considering all the converge/divergence constraints that the elements present. This first solution was inefficient since it has not been capable of inserting all the granule cells, due to their great density value. For this reason a second version has been implemented. It inserts the elements in a *partially random* way following a specific technique which divides the volume in layers and starts by placing firstly the Golgi cells, followed by granules and glomeruli.
Along the z-axis the volume is divided in layers: the user can choose the number of layers. In this case this number is set to five as shown in Figure 2.27. Each layer will contain a number of Golgi cells computed as in Equations 2.27:

$$n\_goc\_z = \frac{n\_goc}{n\_layers\_z} \tag{2.27}$$

At this point, if the remainder of the division is zero, each layer (indicated with the red arrow in Figure 2.27) has to be filled with n_goc_z Golgi cells. Otherwise, the algorithm redistributes the amount of missing cells com-

Element	Density $[1/mm^3]$	Number
Golgi	9000	972
Granule	4×10^6	194400
Glomerulus	3×10^5	32399

Table 2.9: Elements in the network.

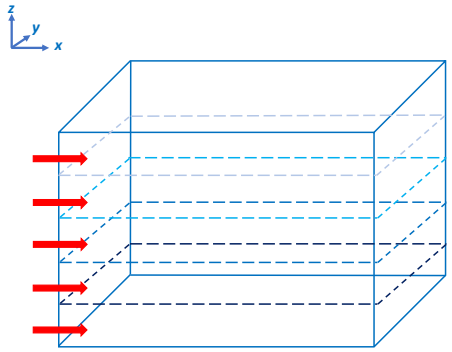


Figure 2.27: *z*-layers in the Golgi cells displacement. The volume is divided in layers in which the Golgi cells will be placed.

puting how many *z-layers* has to contain a number of Golgi cells equal to *n_goc_z*+1 (indicated with *n_goc_z1*). At this point, each *z*-layer (i.e., *x*-*y* plane) is divided in cells which have a rectangular parallelepiped shape and whose length and depth are determined on the base of the soma diameter. In this case, in fact, the cellular soma is represented by a sphere and, in the case of Golgi cell, its diameter is equal to 15 μm [43]. As example, Figure 2.28 presents only the parallelepipeds that fill the first *z*-layer. The

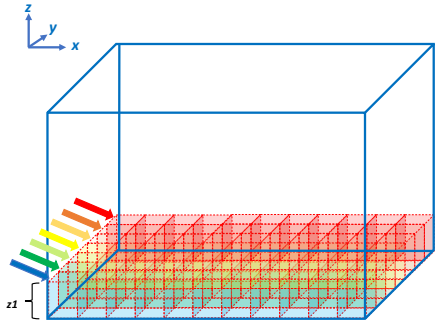


Figure 2.28: Rectangular parallelepipeds in the *x*-*y* plane. Each *z*-layer is divided in rows along the *y*-axis where rectangular parallelepipeds are placed to host the cells.

algorithm begins to place the Golgi cells starting from the blue row of parallelepipeds and scanning all the *y*-rows until the red one is reached. Also in this case, the algorithm computes how many cells has to put in each *y*-row: if the number of cells to put in this *z*-layer was *n_goc_z*, the computation is described by Equation 2.28; otherwise, the Equation 2.29 is used.

$$n\_goc\_y = \frac{n\_goc\_z}{n\_row\_y} \tag{2.28}$$

$$n\_goc\_y1 = \frac{n\_goc\_z1}{n\_row\_y} \tag{2.29}$$

In these Equations, *n_row_y* is the number of rows present in each *z*-layer, that in Figure 2.28 is set to 7 as example. Also in this case, the algorithm evaluates if the remainders are zero and, if not, it will displace *n_goc_y* cells in *row_y_goc* rows and *n_goc_y*+1 cells in *row_y_goc1* rows (in the case of *n_goc_y* to be displaced in a *z*-layer). The same line of reasoning is followed if *n_goc_y1* cells have to be placed and the remainder of Equation 2.29 is not zero. Once determined how many Golgi cells has to be placed in each *y*-row, the algorithm starts the displacement following some rules in order to respect some biological constraints. To better analyze how the algorithm acts, Figure 2.29 shows all the steps to fill a *x*-*y* plane (*z*-layer). Figure 2.29.A shows a top view of the *x*-*y* plane considered before. Each rectangle represents the top face of the parallelepipeds shown before. It is important to highlight that the number of parallelepipeds present in the *x*-axis depends on their size that, in turn, depends on the cellular soma diameter, as previously said. The algorithm begins placing the cells starting from the *y1* layer (light blue color). In this specific example, the algorithm has to place three cells in each *y*-row. The column *x1* and *x13* are inhibited from the cells displacement since the correspondent space is dedicated to the basal dendrites of the Golgi cells, in case the algorithm will place these neurons in boxes *x2* and *x12*. This space is considered only along the *x*-axis (and not along the *y*-one) where the Golgi cell projects its basal dendrites. The space for the apical dendrites is not considered in the granular layer design since they ascend to the molecular layer [43]. In the *y1* row, the algorithm can place the three cells starting from the box *x2* to the *x12*, so it has eleven

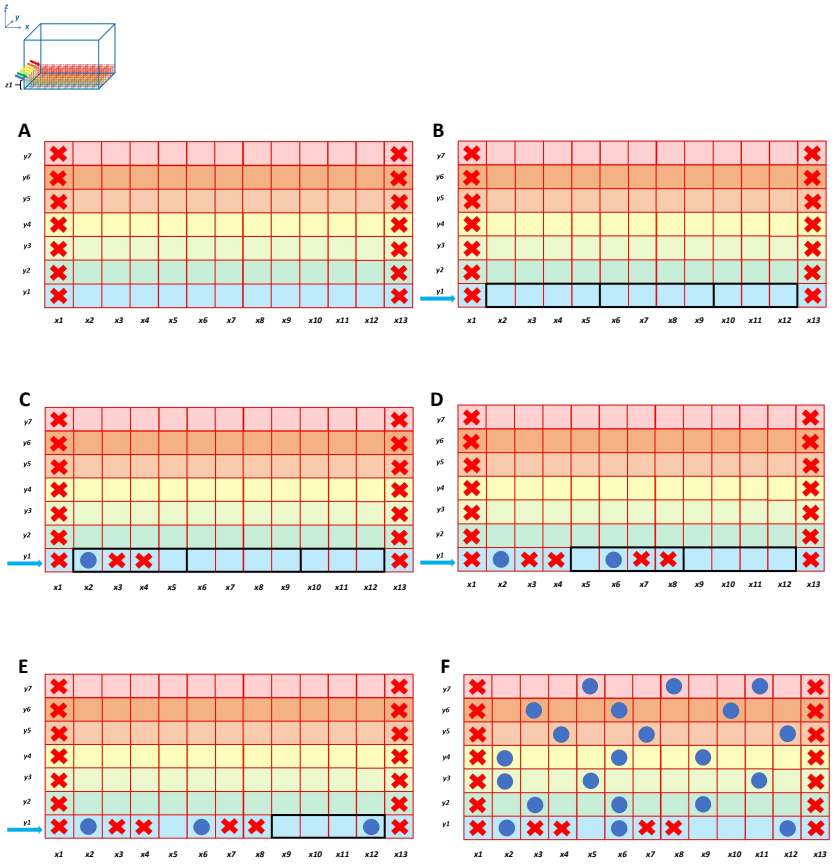


Figure 2.29: Steps to place the Golgi cells in a *x*-*y* plane. A) Top view of a *x*-*y* plane; B) the algorithm considers three sectors to place three cells in the first row; C) one cell is placed and the space for the dendrites is left; D)-E) the second and third cells are placed; F) all the cells are inserted in the first *z*-layer.

boxes in which place the elements. These boxes are divided in three sectors (Figure 2.29.B) where the algorithm randomly inserts a cell: this division

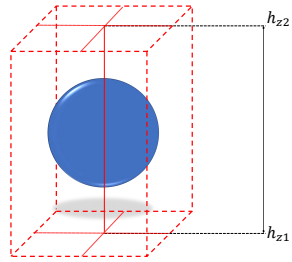


Figure 2.30: Golgi cells into the parallelepiped.

is necessary to avoid that the first element get placed in the lasts boxes of the row. In fact, in this case, it would not have enough space to place the other two cells. Following this technique, the first element is placed in the first area (in this case boxes *x2*-*x5*), for example in box *x2* (Figure 2.29.C). Once the cell is placed, the algorithm inhibits the following two boxes (indicated with the red cross) leaving space for the dendrites. Then, the free boxes start from *x5* to *x12* so this space can be used for the remaining two cells. In this case, two sectors are generated, each containing four boxes (Figure 2.29.D). The second cell is inserted in *x6*, and *x7* and *x8* are left for the dendrites. Finally, when the third cell is placed (Figure 2.29.E), the first row *y1* is completed. In the same way, the algorithm continues to fill the *x*-*y* plane placing three cells per row. Figure 2.29.F shows the entire *z1* layer, with all the cells inserted.
The technique just explained assigns the x and y coordinates to each cell. Let's consider the Figure 2.30 in order to understand how the cell is placed in the parallelepiped and to analyze how the algorithm establishes the z coordinate. The height of the parallelepiped is higher than the soma diameter so that the z coordinate is randomly selected inside the box. For example, in Figure 2.30, the coordinate is chosen in the range h_{z1} and h_{z2}, which are the minimum and the maximum height of the first z-layer (*z1*). In this way, all the Golgi cells inside a layer are not placed at the same height. This technique, just explained for the first z-layer, is applied throughout the volume and, at the end, all the Golgi cells are placed. Figure 2.31.A is a

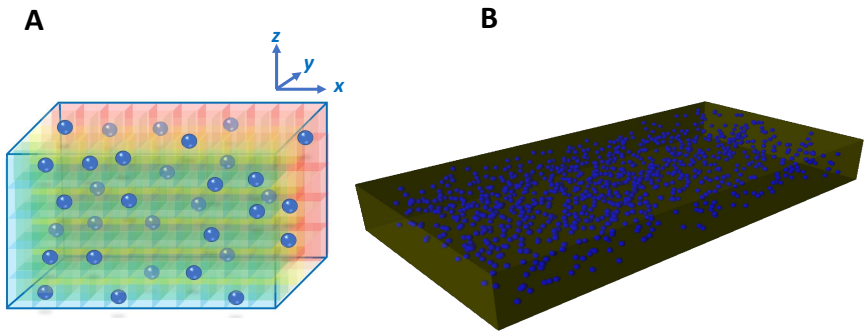


Figure 2.31: Golgi cells displacement. A) Golgi displacement schema; B) Algorithm result.

schema of the Golgi cells displacement while Figure 2.31.B is the algorithm result.

The pseudo-code related to the technique just explained is shown in Algorithm 2. In *line 1* the algorithm evaluates if all the z-layers are characterized by the same number of Golgi cells or not. If not (so the remainder of Equation 2.27 is not zero), two *for* loops have to be executed. The former starts in *line 2* and fills $Zlayer_{mid}$ layers with *n_goc_z* cells; the latter (*line 16*) fills the remaining layers with *n_goc_z1* cells. Inside each *for* loop the *xy*-layers are filled with elements. In *line 3*, the remainder of the Equation 2.28 is evaluated. If it is different from zero, in the y_{th}-xy-plane, a number of rows equal to $XYlayer_{mid}$ has to be filled with *n_goc_y* Golgi cells (*for* loop in *lines 4-6*); while the remaining rows (*for* loop in *lines 7-9*) are filled with *n_goc_y+1* elements. If the *if condition* in *line 3* indicates that all the rows inside a *xy*-plane have the same number of cells, they are all filled with *n_goc_y* cells (*for* loop in *lines 11-13*). The same reasoning is applied in the *for* loop of *lines 16-29* where *n_goc_z1* cells have to be placed: in this case the number of elements to consider in each *xy*-plane is *n_goc_y1*, if

```
1  if z-layers with different number of cells then
2      for z ← 1 to Zlayer_mid do
3          if xy-plane with different number of cells then
4              for y ← 1 to XYlayer_mid do
5                  generate coordinate Golgi cells;
6              end
7              for y ← XYlayer_mid to XYlayer_tot do
8                  generate coordinate Golgi cells;
9              end
10         else
11             for y ← 1 to XYlayer_tot do
12                 generate coordinate Golgi cells;
13             end
14         end
15     end
16     for z ← Zlayer_mid + 1 to Zlayer_tot do
17         if  xy-plane with different number of cells then
18             for y ← 1 to XYlayer_mid do
19                 generate coordinate Golgi cells;
20             end
21             for y ← XYlayer_mid to XYlayer_tot do
22                 generate coordinate Golgi cells;
23             end
24         else
25             for y ← 1 to XYlayer_tot do
26                 generate coordinate Golgi cells;
27             end
28         end
29     end
30 else
31     for z ← 1 to Zlayer_tot do
32         ...
33     end
34 end
```

Algorithm 2: Golgi displacement pseudocode.

```
1  INPUT: n_goc_row, y, z.
2  for j ← 1 to n_goc_row do
3  |   if count = 0 then
4  |   |   min=border;
5  |   else
6  |   |   min=border+box;
7  |   end
8  |   max=(count+1)*(boxes_free/ n_goc_row);
9  |   box=rand()%(max-min+1)+min;
10 |   coo_x= (((box+1)+box)*diam_goc)/2;
11 |   coo_y= (((y+1)+y)*diam_goc)/2;
12 |   coo_z= z_rand* (z_max-z_min+1)+z_min;
13 |   count++;
14 |   store the coordinates in the c_goc array;
15 |   marked as occupied the cubes related to this parallelepiped;
16 end
17 OUTPUT: c_goc.
```

Algorithm 3: Golgi coordinates generation.

the remainder of Equation 2.29 is zero; otherwise, the amount is equal to *n_goc_y1+1*. Finally, if the first *if condition* in *line 1* is false, all the *z*-layers have to be filled with the same number of the Golgi cell (*for* loop in *lines 31-33*). The code of this loop is not shown since it is the same presented in *lines 3-14*. The Algorithm 3 shows the pseudo-code of the function *generateCoordinateGoc* which is called when every row, inside a *xy* plane, has to be filled. This function receives as input the height z of the layer, the depth y of the row inside the *z*-layer and the number of cells to displace in this row (*n_goc_row*). A *for* loop which iterates on *n_goc_row* starts in *line 2* and ends in *line 16*. For each cell, at first the algorithm computes the minimum and the maximum value of the sector in which the cell could be placed. If the algorithm is evaluating the first cell, the minimum is equal to the border dimension; otherwise, it is given by the border added to the box which has been selected for the previous cell. The maximum is computed on the base of the previous selected cell, as said before (Figure 2.29). In *line 9* the box is selected in the just defined range. At this point is possible to compute the x coordinate (*line 10*) as the center of the selected box, which

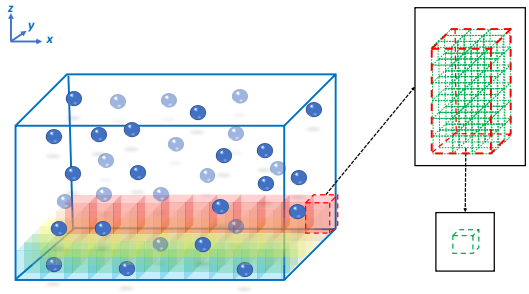


Figure 2.32: Glomeruli and granules cubes.

has a side equals to *diam_goc*, and the y coordinate (*line 11*), always in the center of the box, but determined by the depth of the current row. Then, the z-coordinate is computed as shown in Figure 2.30. The counter is now updated and these coordinates are stored in the *c_goc* array. Finally, the algorithm marks as *occupied* the cubes which are inside the parallelepiped of the Golgi just selected.
Once the Golgi cells are placed, the algorithm has to insert in the volume the glomeruli and the granules. Both are represented by spheres with a diameter equal to 5μm. The strategy adopted to place the glomeruli and the granules is the same of the Golgi cells. In this case, each *Golgi parallelepiped* is filled by several cubes (Figure 2.32), with side equals to 5μm, where the glomeruli and granules will be inserted. During the glomeruli and granules displacement, a further constraint is added to avoid that these elements are placed where the Golgi cells are. When the algorithm has to insert a glomerulus (or a granule), it has to check if the red parallelepiped is already occupied by a Golgi or if it is in the borders, which are inhibited from the displacement, as said before. For this reason, the algorithm has three possible alternatives:

1. the parallelepiped is empty, so it is possible to insert the elements (Figure 2.33.A);

2. the parallelepiped already hosts a Golgi cell, so it is not possible to insert the elements (Figure 2.33.B);

3. the parallelepiped corresponds to the border space, so it is not possible to insert the elements (Figure 2.33.C).

It is important to underline that, if a Golgi occupies a parallelepiped, the cubes inside are inhibited from the displacement in order to leave space for the Golgi dendrites. As explained in Algorithm 3, while the algorithm displaces the Golgi cells, it also marks as *occupied* all the cubes inside its parallelepiped, so that they would not be considered during the glomeruli and granules displacement. As said before, the strategy to fill the volume with glomeruli and granules is the same of the Golgi cells and shown in Algorithms 2 and 3: taking into account how many elements to put in each layer and in each row, it starts from the *z1* layers and places all the cells beginning from the blue rows until the red one. At first, the algorithm inserts the glomeruli and, after having marked these boxes as occupied, places the granules. The result is shown in Figure 2.34. At the end of this phase, the *c_goc*, *c_glo* and *c_grc* arrays are filled with the elements coordinates.
The last step of the *design network* phase is the connection schemes development. They are linear matrices which contain the information of how neurons, glomeruli and mossy fibers are connected following convergence/-divergence rules. The aim of the network is to reproduce the *feedforward* and *feedback* loops: in the first case, the mossy fibers excite the granules and Golgi dendrites and these latter inhibit granules; in the second case,

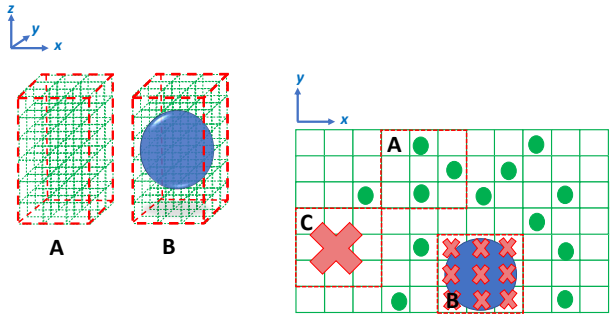


Figure 2.33: Glomeruli and granules displacement.

the mossy fibers excite the granules and, then, the parallel fibers excite the Golgi cells which latter inhibit the granules [44]. The connection schemes generated to reproduce these loops are between:

- granular cells and glomeruli (*link_GrcGlo*);
- Golgi cells axon and glomeruli (*link_GocGlo*);
- Golgi cells basal dendrites and glomeruli (*goc_dend_basal_glom*);
- mossy fibers and glomeruli (*link_MossyGlo*);
- granule (ascending axon and parallel fibers) and Golgi cells (*asc_axon_cluster*, *pf_goc_links*);
- Golgi cells and Golgi cells (*gap_junction*).

Granules - glomeruli connection

According to [35] and [44], the convergence rate between glomeruli and granular cells is 4:1, which means that 3-5 short granule dendrites are con-

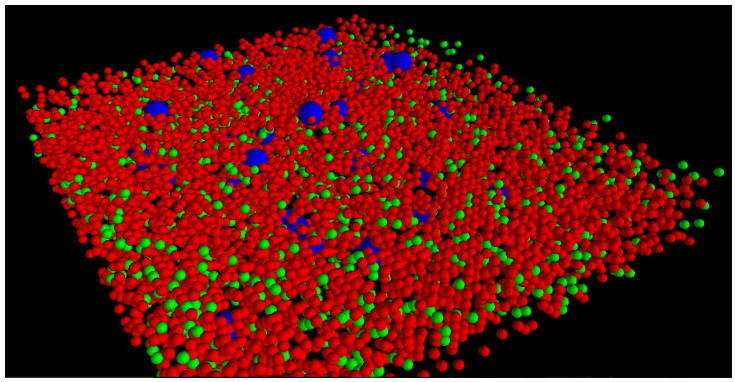


Figure 2.34: Golgi, granules and glomeruli displacement. This image shows the displacement results. In blue there are the Golgi cells, in red the granules and in green the glomeruli.

nected to the same number of glomeruli. The dendrites could not reach glomeruli farther than 40 μm, while the mean of dendritic length is 13.6 μm. Moreover, the granule must project its dendrites to four different glomeruli [35]. The function that develops this connection is *generateLink-GloGrc* and receives as input the granules and glomeruli coordinates and an array *counter_glom*, which takes into account the number of *free position* in each glomerulus. Each glomerulus has about 50 places available for the connection with the granules dendrites, since the divergence rule between a glomerulus and granule dendrites is 1:53 [44]. The aim of this function is to find, for each granule, the four closest glomeruli. The pseudo-code of this function is presented in Algorithm 4. The code starts with the decla-

```
1  INPUT: c_glo, c_grc, *counter_glom;
2  ind_dis structure declaration and initialization;
3  for i ← 1 to n_grc do
4  |   for j ← 1 to n_glo do
5  |   |   if counter_glom[j]!=0 then
6  |   |   |   compute the distance between i and j;
7  |   |   |   if distance <= distance_max_dendrite then
8  |   |   |   |   if distance < last element in the dist array (structure) then
9  |   |   |   |   |   store distance in the last position of the dist array;
10 |   |   |   |   |   store index of the glomerulus in the last position of the
   |   |   |   |   |     index array;
11 |   |   |   |   |   sort the dist elements;
12 |   |   |   |   end
13 |   |   |   end
14 |   |   end
15 |   end
16 |   decrease the number of free positions in the glomeruli just linked to the
   |     granules;
17 end
18 store the connections in the link_GrcGlo arrays.
19 OUTPUT: link_GrcGlo.
```

Algorithm 4: Granule - glomeruli connection.

ration of the structure *ind_dis* which is made up of two arrays, each with four positions: *dist* contains the distances between the elements and the *index* the glomeruli indexes. In *lines 3-4* two *for* loops start: the former

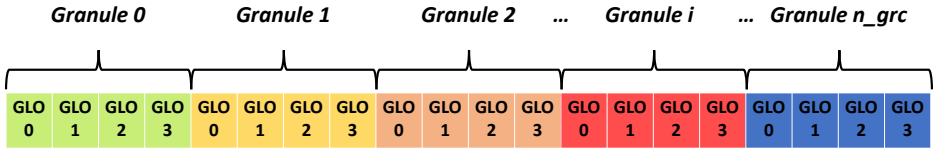


Figure 2.35: ***link_GrcGlo*** **array.**

iterates on the granules number and the latter on the glomeruli. If the j-th glomerulus has at least one place available for the connection (*line 5*), the Euclidean distance between the i-th granule and the j-th glomerulus is computed. If this distance is lower than the maximum dendrite length (i.e., 40 μm) (*line 7*), it can be considered for the connection. Moreover, if this distance is lower than the one stored in the last position of the *dist* array (*line 8*), it is overwritten in the same position of the *dist* array and also the glomerulus index j is stored in the *index* array (*lines 9-10*). The *dist* elements are, then, sorted in ascending order, keeping track of the correspondent indexes. When four glomeruli have been selected for a granule, the counter which takes into account the free position in the correspondent glomeruli is decreased. At the end, the connections are stored in the linear matrix *link_GrcGlo*, which presents the elements as shown in Figure 2.35. This array contains a number of elements equal to the number of connections per each granule (that is four) multiplied by the granules number. For each granule, the indexes of the four connected glomeruli are stored.

Golgi (axon) - glomeruli connection

The Golgi axons are placed in the granular layer spreading longitudinally. They enter in the glomeruli to inhibit the granule cells. The convergence rate between the glomeruli and the Golgi cell is 50:1 [35] [44]. In this case, a crucial rule has to be followed: a Golgi axon can enter only in glomeruli without granule cells in common. In this way a granule cell is not inhibited twice by the same Golgi cell. The function that generates this

connection is called *generateLinkGloGocAxon*. It generates the connection in the same way seen in Algorithm 4, with a further step. It compares the Golgi cells and the glomeruli: if their distances are less than 150 μm (the mean of the axon length), the value is considered for a possible connection. At this point, if the distance constraint is satisfied, the algorithm has two choices: 1) if it is evaluating the first glomerulus for that Golgi cell, the glomerulus index is stored; 2) otherwise, it has to check if the selected glomerulus hosts some dendrites belonging to granules which, in turn, are hosted in glomeruli already selected for that Golgi cell. If this check is passed, the glomerulus index is stored for that Golgi cell. The array which contains all the Golgi axons - glomeruli connection is *link_GocGlo*, and follows the same strategy described in Figure 2.35: for each Golgi cell, the related glomeruli indexes are stored.

Golgi (basal dendrites) - glomeruli connection

The Golgi basal dendrites spread around the soma. They reach the glomeruli where they make excitatory synapses with the mossy fibers. The number of glomeruli that the basal dendrites of a single Golgi can reach is computed as follows:

$$n\_glo\_per\_dend = \frac{n\_glo}{n\_goc} \tag{2.30}$$

Since the basal dendrites are placed in the bottom part of the Golgi soma, the algorithm checks the glomeruli which are characterized by a z coordinate which is lower than the Golgi one. Moreover, since the basal dendrites spread along the x axis (as said before in the Golgi displacement), the algorithm searches the glomeruli in an elliptic cylinder, that is a cylinder with an elliptical cross section. In this case the major axis is determined by the maximum length of a basal dendrite, while the minor axis is obtained by the NEURON simulator settings. The Golgi soma is in the center of the top elliptical face of the cylinder. In this case, the function that generates the connections is called *generateLinkGloGocBasalDendrite* and compares the Golgi cells with glomeruli. If a glomerulus is inside the elliptic cylinder,

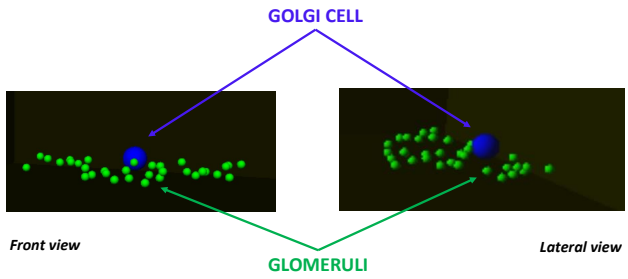


Figure 2.36: Golgi basal dendrites - glomeruli connections. The image shows in blue the Golgi soma and, in green, the glomeruli connected to the Golgi cell through the basal dendrites.

built as just explained around the Golgi soma, its index is stored in a suitable array, called *goc_dend_basal_glom*. In Figure 2.36 it is possible to see the glomeruli chosen by the algorithm for a specific Golgi cell.

Mossy fibers - glomeruli connection

The main input to the cerebellum comes through mossy fibers, which enter in the granular layer and branch longitudinally generating numerous *rosettes*, i.e. clusters of glomeruli [44]. In order to reproduce this input, at first the number of mossy fibers that reaches this portion of the network has been computed as the ratio between the number of glomeruli and the number of ramifications that a mossy shows. Once the number of mossy is known, the algorithm starts to link them to glomeruli: since they branches along the x-axis, the volume is divided in *sectors* (as shown in Figure 2.37) and each group of mossy fibers reaches one of this sectors. To link a mossy fiber to a glomerulus, the algorithm firstly sorts all the glomeruli in ascending order accordingly to their y coordinate. Then, it defines the sectors whose minimum and maximum are represented by the glomeruli with the lowest and the highest y and x coordinates, respectively. Inside a sector, a defined number of mossy fibers (three in Figure 2.37) choose the glomeruli

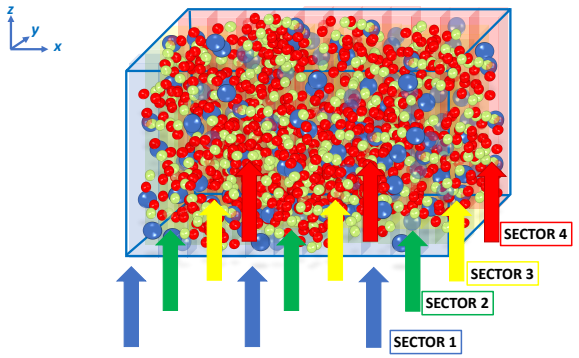


Figure 2.37: Golgi basal dendrites - glomeruli connections. The image shows in blue the Golgi soma and, in green, the glomeruli connected to the Golgi cell through the basal dendrites.

in a random way. At the end, all the connections are stored in the linear matrix *link_MossyGlo* which contains the glomeruli indexes for all the mossy fibers.

Granule (ascending axon and parallel fibers) - Golgi cells connections

The granules axons cross vertically the cerebellar *purkinje layer*, which contains the Purkinje soma, and reach the *molecular layer* where it branches into parallel fibers which run trasversally, i.e., along the y axis. Even if this work aims to reproduce the granular layer, it is important to take into account these connection schemes in order to reproduce the feedback and feedfarward loops, simulating all the connections between neurons. It has been observed that granule cells form their connections through parallel fibers but also along the ascending axon, so the part of axon which pass through the purkinje layer before branching [43]. Moreover, work in [44] reports that the convergence rate between ascending axon and Golgi cell is 400:1 and between parallel fiber and Golgi cells is 1000:1. Firstly, the algorithm computes the connections between the ascending axons and the Golgi cells. The idea is to create a cluster which surrounds the Golgi cell,

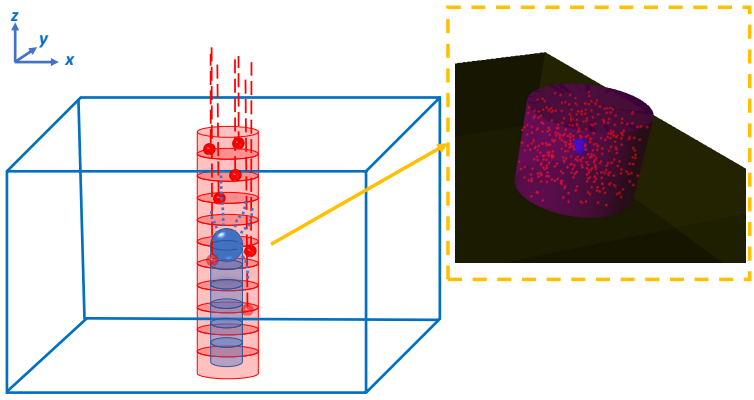


Figure 2.38: Granular ascending axon - Golgi connection. The image shows in blue the Golgi soma and, in red, the granules. The elliptic cylinder in light red is the space where the granules are selected for the connections. The red dotted lines represent the acending axon while the blue dotted lines are the Golgi apical dendrites. In the orange box the algorithm result is shown: the red spheres are the granules selected for the Golgi cells in blue.

where the algorithm can choose granule cells to link. The building of this cluster is shown in Figure 2.38. The blue sphere is the Golgi soma and around it the elliptic cylinder cluster (in red) is built. The major axis of the ellipse is defined by the maximum length of the apical dendrites, while the minor is given by their depth. For each Golgi, the algorithm selects 400 granules inside this cluster (red cylinder) avoiding the blue cylinder which is the area under the Golgi soma, where is less possible to have connections. The algorithm, thus, randomly selects a granule and checks if it is inside the red cylinder. If yes, its index is stored in the linear matrix *asc_axon_cluster* which contains all the granules indexes per each Golgi cell. As far as the parallel fibers are concerned, the algorithm evaluates two connection typologies: the *distal* and *local* parallel fibers. The Algorithm 5 presents how the code computes these connections. The inputs of the algorithm are the number of distal and local connections (*n_connectionL*, *n_connectionD*) and the granules and Golgi coordinates. In *line 2* a *for* loop, which iterates on the Golgi cells, starts. The algorithm evaluates *z_dend_tree* which is the

```
1  INPUTS: n_connectionL, n_connectionD, c_grc, c_goc;
2  for i ← 1 to n_goc do
3  |  compute z_dend_tree, the maximum height of the apical dendritic tree;
4  |  for j ← 1 to n_grc do
5  |  |  if counter_local=num_connectionL then
6  |  |  |  if j-th grc is inside the ellips then
7  |  |  |  |  if z_coord + aa_height <= z_dend_tree then
8  |  |  |  |  |  store grc index and computes the time;
9  |  |  |  |  |  counter_local++;
10 |  |  |  |  end
11 |  |  |  end
12 |  |  end
13 |  |  if counter_distal=num_connectionD then
14 |  |  |  if j-th grc is inside the parallelepiped then
15 |  |  |  |  store grc index and computes the time;
16 |  |  |  |  counter_distal++;
17 |  |  |  end
18 |  |  end
19 |  end
20 end
21 OUTPUT: pf_goc_links.
```

Algorithm 5: Parallel fibers - Golgi connection.

maximum height of the apical dendritic tree (*line 3*). It is important to know this data in order to understand if the parallel fibers will cross the dendritic tree, and do the connection, or not. Then the granules are evaluated (*line 4*): at first the algorithm checks if all the local connections have been performed (*line 5*). If not, it evaluates if the j-th granule is inside an ellipse built around the Golgi cell (*line 6*), presented below. If yes, the algorithm computes if the parallel fiber crosses the Golgi dendritic tree (*line 7*): this means that it has to check if the sum of the z-coordinate of the j-th granule and its ascending axon is lower then *z_dend_tree*. If yes, the connection is generated. Then, the algorithm continues searching the distal connections with the same strategy (*lines 13-18*). The main difference is that in this case it is checked if the granule is inside a parallelepiped and not an ellipse, as shown below. It is important to underline that, in the

case of the parallel fibers connection, are stored both the granules indexes and also the time that the signal takes to start from the granule, to pass through ascending axon and parallel fiber and reach the Golgi dendrite. The time is computed by dividing the covered distance by the signal velocity. In this phase of the work, the distance value is approximate and it is given by the sum of the ascending axon length and the distance between the Golgi and granule soma. This is an approximation because in this case these two terms are considered as straight lines while in nature they are not. A more realistic distance computation will be included in the future. Moreover, since the divergence rate between parallel fiber and Golgi is 1:1.9 [44], a parallel fiber is connected twice to a granule. For this reason two instants of time are stored, representing both the connections. During the time computation a random term is inserted in order to represent the two connections at two different points along the parallel fiber. Figure 2.39 shows the ellipse and the parallelepiped used to compute the local and distal connections. Finally, it is important to notice that if the simulated volume is too small to find distal parallel fibers the algorithm will automatically select only the local parallel fibers for the connections.

Golgi cells - Golgi cells through gap juctions

Golgi cells are connected through gap-junctions present in the apical dendrites [44] [43], as shown in Figure 2.40.

In this case the algorithm, for each Golgi cell, selects the two closest Golgi cells to perform the gap junction. When the dendritic tree morphology will be inserted in the model, a more realistic connection will be studied.
Once the connection scheme have been generated, the network activity can be simulated.

2.7.2 Granular layer simulators

In this phase of the work, it is possible to simulate the network activity using the neuronal simulators and the connection schemes presented in

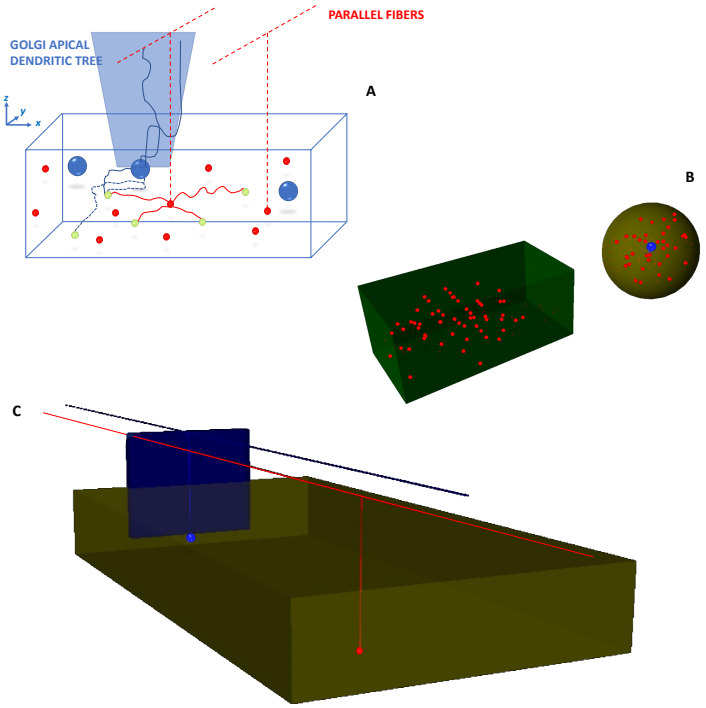


Figure 2.39: Parallel fibers - Golgi connections. A) Scheme that shows the connection between a parallel fiber (red dotted line) and a Golgi cell apical dendrite, inside the dendrtitic tree (in light blue); B) granules selection for the local and distal connections; C) link between a parallel fiber and a Golgi generated by the model.

this chapter. In fact, once the Golgi and granule cells activities are reproduced, their outputs can be correctly exchanged between the granular layer elements on the base of the connections.

The serial algorithm

Figure 2.41 shows the flow of the network serial algorithm. The *First phase* of the algorithm consists in the connection schemes reading. Moreover, some schemes have been joined in order to facilitate the signal ex-

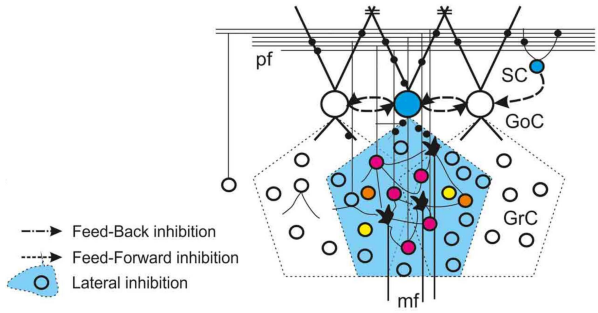


Figure 2.40: Golgi connections through gap junctions. The image shows the connection of Golgi cells through gap junctions (indicated by =) in the apical dendrites. In the image *GoC* are the Golgi cells, *pf* the parallel fibers, *GrC* the granule cells and *mf* the mossy fibers [43].

change during the activity simulation. For example, mossy fibers excite granules in the glomeruli. For this reason, the scheme *link_MossyGlo*, containing the mossy - glomeruli connections, has been joined with the scheme *link_GrcGlo*, containing the granule-glomeruli connections. The new scheme is called *link_mossy_grc* which directly link the mossy fibers to the granule cells. During the *Second phase* the granule and Golgi cells parameters are declared and initialized, as previously described. The *Third phase* concerns the computation of the spikes queues produced by the mossy fibers. In fact, through the *SpikeTrainGenerator* function, presented above, a spike queue for each mossy is created. Then, these spikes are stored in the *spike_queue* arrays contained in the synapses of the Golgi and granule cells connected to those mossy. Then, the algorithm can start the network activity computation. What it is important to notice is that: while the cellular simulators analysed, for each cell, its whole activity, the network simulator has to evaluate the same instant of time for all the cells. The reason is that in the first case the cells were independent, so the algorithm could compute the entire activity of a cell and then analyse the following one. Now neurons have to communicate so, after evaluating a single instant time of a

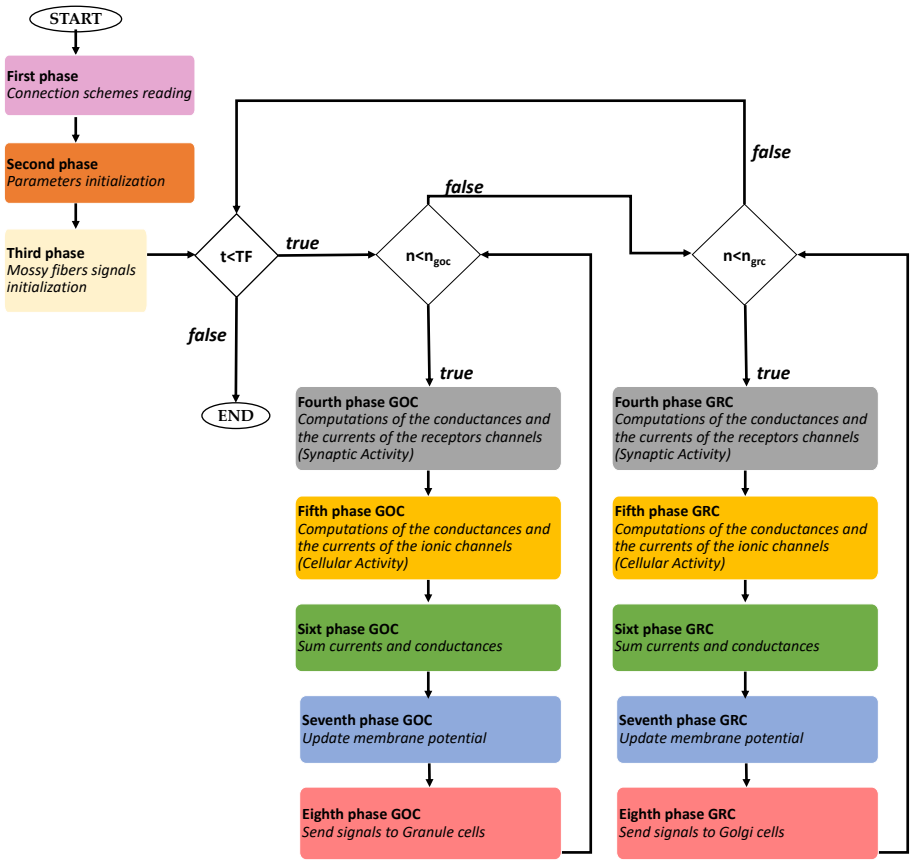


Figure 2.41: Granular layer network - serial algorithm.

cell, the generated signal is transmitted to the connected element. For this reason, the first *for* loop iterates on the time of the simulation. Then, the t-th instant of time of the Golgi cell activity is evaluated as shown before: the algorithm analyses the synaptic and cellular activity, sums all the currents and updates the membrane potential (*Fourth - Seventh phases GOC*).

Moreover, it evaluates if an action potential has occurred and, if so, sends this signal to the granules connected to the specific Golgi (*Eighth phase GOC*), according to the connection schemes. In particular, here there is the inhibitory connection between the Golgi axons and the granule dendrites which is performed in the glomeruli. For these reason the connection schemes *link_GocGlo* and *link_GrcGlo* are used. Once the *t*-th instant of all the Golgi cells has been computed, the algorithm evaluates the same instant of time of all the granules. Then, the granules activity is analysed, as described above (*Fourth - Seventh phases GRC*) and, if a granule generated a spike, it will be sent to the Golgi cells linked through ascending axons or parallel fibers (*Eighth phase GRC*). In this case the connection schemes *asc_axon_cluster* and *pf_goc_links* are used to correctly send the signals. As the granules computation ends, the algorithm evaluates the following instant of time.

The parallel algorithm

The parallel algorithm (Figure 2.42) starts on the host where the first three phases, already explained for the serial version, are performed. As in the previous parallel codes, also in this case the data have to be prepared before the transfers (*Vector packaging* in the *Second phase*). The network activity computation begins on the host where a *for* loop, which iterates on the time steps, is performed. In the *Fourth phase* the algorithm analyses if the first element of each mossy fibers spikes queue is lower than the current time, i.e. the *t*-th step. The evaluation of all the mossy queues is performed by a *for* loop which is parallelized using the OpenMP API, exploiting the *#pragma* statement. To notify that the mossy fibers has generated a spike to be considered in this step, a new strategy has been developed. Two arrays called *goc_spike* and *grc_spike* are allocated. They contain integers variables and their sizes are equal to the number of Golgi and granules cells, respectively. Let consider, for example, the mossy fiber with index 10, which generates a spike in the current time step. This mossy fiber excites the granule 200 and the Golgi 300: in this case, the 200-th element of the *grc_spike* array and the 300-th element of the *goc_spike* array

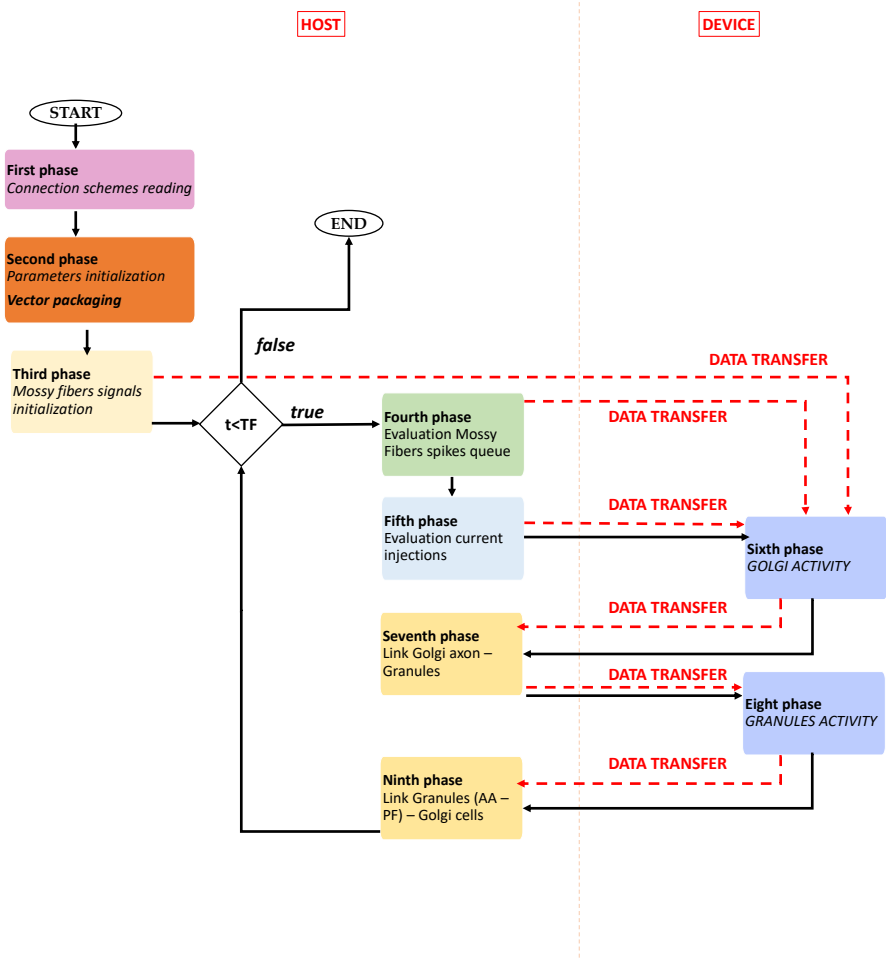


Figure 2.42: Granular layer network - parallel algorithm.

are set to 1. These arrays are re-initialized and updated at each *for* loop iteration. Moreover, they are transferred from host to device in order to be evaluated during the Golgi and granules synaptic activity computation. The algorithm proceeds evaluating if some cells are stimulated by injected currents: if yes, the current amplitude is trasferred from host to device.

At this point, the host launches a kernel which computes the current time step of the Golgi cells activity. This phase is performed by a kernel since the threads evaluates the Golgi cells activity in parallel: in fact, each time step of the neuronal activity is independent from the one of the other cells. The Golgi cell activity code is the same presented in the parallel simulator above. As the *Sixth phase* is completed, an array containing the membrane potential of all the Golgi cells is transferred from device to host. Here, these potentials are evaluated (*Seventh phase*) and, if a Golgi has generated a spike, the connected granule cell has to consider it in its activity. For this reason, in the *grc_spikeINIB* array (whose size is equal to the granules number), the flag corresponding to the granule cell, linked to the specific Golgi, is set to 1. This array is transferred to the device, where a kernel computes the granule cells activity in the current step time (*Eight phase*). An array containing all the granules membrane potential is transferred from device to host. Here (*Ninth phase*), the algorithm evaluates if the granule cells generated a spike. If yes, the Golgi cells linked to the specific granule (through ascending axon and parallel fibers) has to be notified. The modality is the same used before: if a Golgi is stimulated by the granule input, its flag in the array *goc_times_spikeAPIC* (whose size is equal to Golgi cells number) is set to 1. This array is then transferred to the device to start the computation of a new time step.

2.8 Results

In this chapter, the development of the granular layer network has been presented. At first, the Golgi and granular cell simulators have been implemented evaluating different technologies and programming strategies. As described before, in order to realistically reproduce the neuronal functionalities, complex models have been used in the algorithms development. Among the technologies evaluated, the best performing is the GPU which significantly decreases the elaboration times of the serial code and of the multicore solution. The neuronal simulators have been included in the granular layer network development. This phase of the work consisted in

two main steps: the network design and simulation. The aim was to reproduce a cerebellar volume with dimensions 600×150×1200 μm, which are the values indicated by the Neurophysiology laboratory of the University of Pavia. In fact, this network configuration can reproduce the granular layer functionalities, allowing the physiological validation. The algorithm is able to place 972/972 Golgi cells, 194,333/194,400 granules, 32,399/32,399 glomeruli and 1080 mossy fibers and to connect these elements in about 13 minutes (758.691 s), exploiting the CPU present in *System 1*. Moreover, the serial version of the network simulation takes about 35 hours (125,056.78 s) in reproducing 3 s of neuronal activity, if the same system is used. It is clear that this huge amount of cells, which communicate each others, demands a more powerful technology in order to reduce the computational time of the network activity reproduction. For this reason, the parallel version has been developed and tested using the Tesla K40 GPU of *System 1*. In this case, 3 s of the network activity are reproduced in about 7.5 hours (27,100.52 s) obtaining a significant reduction in the elaboration time. Once more, the GPU device provides a faster result that reflects the efficiency of this technology in the computational modelling problems.
Despite this important result, some aspects have to be highlighted: these times are far from the real-time simulation which, probably, could be reached only using very powerful supercomputers. Anyway, in this thesis it has been demonstrated how the multi-GPU systems could help in the computational time reduction. Therefore, one of the first future works will be the parallel algorithm development to exploit both the Tesla K40 GPUs present in *System 1* and also the Galileo supercomputer resources. Moreover, it could be interesting to evaluate how the computational time varies when the network dimensions increase, even if this will not modify the significance of the model implementation. Even if the network dimensions used in this work are enough to reproduce the main behaviours of the granular layer, a bigger network will represent a further step into reproducing the whole cerebellum. Evidently, the use of supercomputers will be mandatory in the simulation of very large networks both for memory occupancy and computational power demand.
Concerning the neuronal activity, this network can reproduce the physio-

logical cellular behaviors when Golgi and granules cells are stimulated by current injections. The next functionalities and characteristics to be validated are the center surround organization, high-pass filtering in responses to spike bursts and the coherent oscillations in response to diffuse random activity [35]. Finally, a more detailed neuron morphology will be introduced.

At the state of the art, several neuronal networks have been developed. The main difference with the one presented in this work, is the mathematical models used in the neurons description. During this thesis, it has been explained the importance to choose the suitable model for the application goals. The aim of this work was to build a cerebellar network capable of realistically reproducing the granular layer behavior. For this reason, the used models were very complex and heavy to solve. In [45], for example, authors developed an event- and time-driven spiking neural network simulator for an hybrid CPU-GPU platform. It consists of a very dense granular layer and a Purkinje layer with a small number of cells. In this work, neurons are reproduced using LIF models and characterization tables (computed offline) which contain the dynamic of each cell. Authors test several configurations varying parameters as the cells number and the integration steps. The best result obtained is the simulation of 3 million neurons and 274 million synapses using 32 GB CPU RAM and 1.28 GB GPU RAM. Reproducing 10 s of neuronal activity takes 987.44 s. This result can not be directly compared to the elaboration times obtained in this work for two main reasons: the most important is the different model chosen and the other is related to the integration step. It is clear that in this thesis the LIF model could not be considered due to the aim of the work, widely explained. As far as concerned the integration step, the one used in [45] varies from 0.1 to 1 ms which is higher than the one used in this work, which is 0.025 ms. This characteristic significantly increases the computational time since the model equations have to be solved a greater number of times.

Another interesting work, which reproduces the cat cerebellum network containing more than a billion spiking neurons, is described in [46]. Authors used 1280 PEZY-SC processors to elaborate in real-time 1 second of

neuronal activity, with an integration step of 1 ms. Also in this case, cells are described by LIF models and the connectivity rules are not updated. Moreover, the synapses are characterized only by the AMPA receptors. Finally, this work presents several different aspects which do not allow a fair comparison.
Authors in [47] provides a tool able to build, visualize and analyze network models in 3D space. The network design is capable of reproducing very realistic and complex neuron morphologies and the mathematical models are based on the Hodgkin and Huxley one. Nevertheless, they run simulations of up to only 5,000 neurons on a single-processor machine, which take 1-2 hours for 4 seconds of activity. In this case, even if the morphology included in this work is very detailed, the simulation part is not so efficient as the one proposed in this work. The computational time of this elaboration is not very fast if the number of reproduced cells is considered.
On the other hand, the cerebellar granular layer network developed in [35] is the one considered as reference of the present work. In fact, these networks present the same mathematical models (even if their models are written for the NEURON simulator) and connection rules. The main difference concerns the cellular morphology and the elements displacement. In this case, the cellular soma is represented by a point (and not sphere) and this means that two soma can be overlapped, aspect that is not realistic. Moreover, during the cells displacement, the algorithm does not take into account the minimum distances between cells. In this case, they create a network inside a 3D space (i.e., a cube with 100 μm edge length) and which includes 315 mossy fibers and 4,393 neurons (4,096 granules, 27 Golgi cells and 270 basket and stellate cells). The reproduction of 3 s neuronal activity requires about 20 h on a Pentium-5 dual-core and 30 minutes using 80 CPUs on the CASPUR parallel cluster.
It is possible to conclude that the granular layer simulator described in this work is characterized by a high level of realism due to the mathematical model and the connection rules used. Moreover, the use of HPC technologies has allowed to significantly reduce the computational time of the cellular activity reproduction of a great number of neurons.

Chapter 3

Single and *multi*-GPU processing for brain cancer detection exploiting hyperspectral imaging

Several causes make the brain cancer identification a challenging task for neurosurgeons during the medical operation. The surgeons' naked eye sometimes is not enough to accurately delineate the tumor location and extension. For this reason, a support system which provides a real-time accurated cancer delimitation is essential in order to improve the surgery success.
The system described in the following paragraphs meets these requirements exploiting a non-invasive technique suitable for medical diagnosis: the hyperspectral images.

3.1 Introduction

The surgery, together with the radiotherapy and the chemiotherapy, is one of the most efficient treatment solutions for the brain tumor. The main issue in the brain cancer identification is its diffusion and infiltration in the surrounding healthy tissues that make its individuation a difficult task for the surgeons' naked eye. It is crucial, thought, an extreme accuracy and precision in the cancer resection [48] [49]. As a matter of fact, the majority of tumour recurrences are often linked to the presence of cancerous regions unintentionally left during the surgery. On the other side, it is also very important to not over-resect the brain tissue in order to not remove the healthy brain parts to not cause permanent neurological deficts in the patient [50]. Today several support systems help the neurosurgeons in the tumor localization but they still present some limits that preclude the obtaining of a highly accurate identification. The neuro-navigation, for example, is highly influenced by the brain shift and the tumor changes during the resection that prevent an accurated delineation of the cancer borders [51]. To overcome this kind of problem the intra-operative Magnetic Resonance Imaging has been adopted even if it is very expensive, it has poor resolution and it significantly increases the surgery duration [52]. Finally, the 5-aminolevulinic (5-ALA) fluorescent tumor markers are able to identify high grade tumors even if they are not very accurate in the margins delimitations [53]. To overcome these limitations, a label-free, non-ionizing, non-contact and non-invasive technique as the Hyperspectral Imaging (HSI) is adopted. It is a form of imaging spectroscopy that produces a three-dimensional image whose pixels are characterized by spectral information of the acquired scene. This cube contains the reflectance values of the brain surface, so the fraction of incident electromagnetic radiation upon a surface that is reflected. This value changes according to the wavelength distribution of the incident radiation. The hyperspectral images provide more information than the RGB ones since they are characterized by a large number of contiguous and narrow spectral bands over a wide spectral range. The characteristic that makes the HSI a valid solution for the brain identification is that it provides the pixels *spectral signature* that

can identify the type of tissues. In fact, the spectral signature is given by the variation of the reflectance and emittance values of the tissue with respect to the wavelengths. These values depend on the tissue chemical composition, on its physiological behaviour and on its temperature. Since the healthy and the cancerous tissues have different physiological characteristics, they also have different spectral signatures. A system able to discriminate the spectral signatures of the tissues present in the captured scene could be an useful support system for the neurosurgeons [54]. A crucial specification that the system has to satisfy is a real-time elaboration in order to not slow down the medical operation. Due to the heavy computations that the system has to perform, the use of the High Performance Computing (HPC) is mandatory to achieve real-time compliancy. The development of this system is part of the HELICoiD (HyperEspectraL Imaging Cancer Detection) European project funded by the Research Executive Agency (REA). It is led by the Universidad de Las Palmas de Gran Canaria (ULPGC) and involves four univerisites, two universities hospital and three leading industry partners. Therefore, the HELICoiD brain cancer detection system goal is to support neurosurgeons during the tumor resection, providing a map of the brain where different tissues are displayed with different colors. In this way, the surgeons are facilitated in the cancer individuation and delimitation.

3.2 State of the art of hyperspectral imaging in medicine

Originally the hyperspectral images were used in remote-sensing in military area. As the years go by, they become very useful in different fields such as in the geological, archaeological and aerospace ones. They were used only by few companies and research insitutes since the hyperspectral cameras and the computational systems were very expensive. The recent technological progress has allowed a more widespread use of the hyperspectral images also in other fields, such as medicine and, in particular, cancer detection. The fact that each material or tissue differently reacts to the incident radiation on his surface, on the base of its molecular structure, can

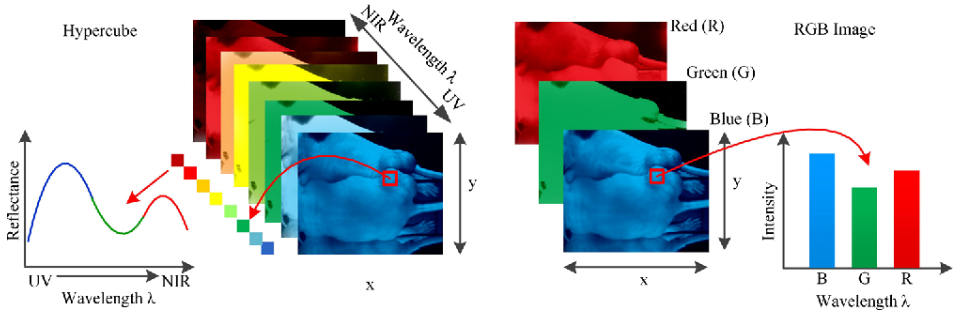


Figure 3.1: Comparison between an hyperspectral cube and a RGB image. The image shows the main differences between the hyperspectral and the RGB images. In particular, it is possible to notice the continuous reflectance curve of a pixel of the hyperspectral image (on the left) and the discrete intensity curve of a pixel of the RGB image (on the right) [54].

be exploited in the discrimination between healthy and tumor tissue. In an hyperspectral image, this reaction is visible in the pixel electromagnetic spectrum which provides detailed information about the material typology of the corresponding area that the pixel is representing. The hyperspectral image, also called *hypercube*, is a three-dimensional dataset of a two-dimensional image on each wavelength [54]. It is characterized by a number of bands higher than the standard RGB images, since they have only three bands (one for the red channel, one for the green and one for the blue). The hyperspectral images are characterized by hundreds of bands per pixel and this issue allows to have a spectral signature of each pixel with the refelctance value in each band. In Figure 3.1, it is possible to notice the differences between these two kinds of images and, in particular, the comparison between the *reflectance curve* (or spectral signature) provided by the hypercube and the RGB intensity curve [54]. As said before, recently the HSI is often used in screening, detection and diagnosis of several diseases such as heart and circulatory pathologies, retinal diseases, diabetic foot, shock and different types of cancer. The reason why this technique is exploited especially in the cancer detection is that the biochemical and morfological changes associated with lesions modify the optical characteristics of a tissue, such as the absorption, scattering and the fluorescence. These

variations provide valuable diagnostic information useful in the diagnosis and detection phases. Another technique used in this field is the optical spectroscopy but the HSI is able to capture more extended images and provides more accurate results in the detecting cervix, breast [55], skin [56], ovaries [57], prostate [58], tongue [59] and brain cancer. The main differences between these cases of study are the acquisition system setup, the nature of the samples (in-vivo, ex-vivo, in-vitro), the studied disease [48] and the adopted classification algorithms [54]. In the HELICoiD project the disease under study is the *glioblastoma*, a malignant tumor affecting the nervous system which usually occurs in the adult patients. Nowadays the brain cancers are among the most common tumors and their incidence is about 3.4 per 100,000 subjects. The *glioma* is the tumor which affects the glial cells of the brain and it is one of the most common form of brain cancer. It represents the 30-50% of the primary brain tumors and in the 85% of these cases this tumor occurs in the form of glioblastoma, which is locally very aggressive, in most cases unicentric and rarely metastasizing. This type of glioma is also characterized by a fast-growing invasiveness [60] [61]. Traditional diagnosis of this type of tumors are provided by the information generated by the *excisional biopsy*, which consists in a surgical cut to resect the tissue from the living body, followed by histology or citology. This diagnostic methodology is very invasive with potential side effects due to the over-resection of the tissue (removing also healthy tissue) or due to the fact that some cancerous parts can be left in the patient. Moreover, this methodology is not a real-time diagnosis since the tissue samples have to be analyzed in the laboratory [61]. The system, developed in this project and optimized in this work, tries to overcome these limitations.

3.3 The hyperspectral cameras and the images acquisition for the database

As previoulsy said, the HELICoiD brain cancer detection system provides a brain map where the tumor is hilighted in order to help the neurosurgeon to resect it. An overview of the complete system is shown in Figure

3.2. The system is made up of an *Acquisition Scanning Platform*, which captures the hyperspectral image, a *Data Pre-processing System* where the HS cube is pre-processed following the modalities explained below, a *Processing Sub-system Platform* where the pre-processed image is classifyed exploiting different classification algorithms. Furthermore, an *User Interface* is provided. The hyperspectral acquisition system (Figure 3.3.A) [63] [64] is composed of two cameras: the Visual and Near Infra-Red (VNIR) and the Near Infra-Red (NIR). The camera used in the study is the Hyperspec® VNIR A-Series, provided by the Headwall Photonics Inc. (Massachusetts, USA). This camera covers the spectral range from 400 to 1000 nm (visible and near infrared frequency), capturing 826 spectral bands, with a spectral resolution of 2-3 nm and a pixel dispersion of 0.74 nm. This device integrates a Silicon CCD detector array with a minimum frame rate of 90 fps, where the frame in this case is a line of 1004 pixels and 826 spectral bands [48]. This camera exploits the *push broom* technique in the image acquisition: at first, the 2-D matrix of the camera captures the complete spectral dimension of the first image spatial dimension of a scene. Once the acquisition is completed, the camera field of view (FOV) relative to the scene is shifted so that the second spatial dimension can be acquired. Usually, to perform the *push broom* scanning technique, the sensors array is moved. For this reason, the camera is shifted using a stepper motor connected to the processing

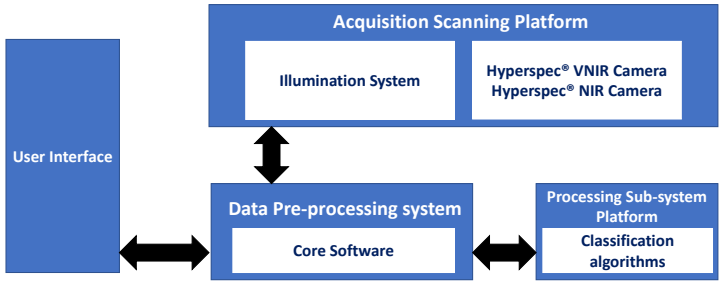


Figure 3.2: HELICoiD system overview [62].

system through a RS-232 port. The Acquisition Scanning Platform contains also an illumination device (Quartz Tungsten Halogen (QTH) lamp) capable of emitting cold light in the range between 400 and 2200 nm. This kind of lamps generates high temperatures that could cause damages to the patient. To preserve the brain surface from these temperatures, the lamp is connected to the cold light emitter through an optical fiber [63]. In Figure 3.3.B the hyperspectral acquisition system is shown while capturing a brain image during a neurosurgery routine at the University Hospital Doctor Negrin of Las Palmas de Gran Canaria (Spain) [63]. In this work, a set of five in-vivo brain surface hyperspectral images has been acquired during four neurosugeries taken place in the University Hospital Doctor Negrin of Las Palmas de Gran Canaria. The four patients presented a grade IV glioblastoma tumor confirmed by histopathology. The study protocol and consent procedures were approved by the *Comité Ético de Investigación Clínica-Comité de Ética en la Investigación* (CEIC/CEI) of University Hospital Doctor Negrin and written informed consent was obtained from all subjects [63]. A standard procedure has been followed in order to acquire the

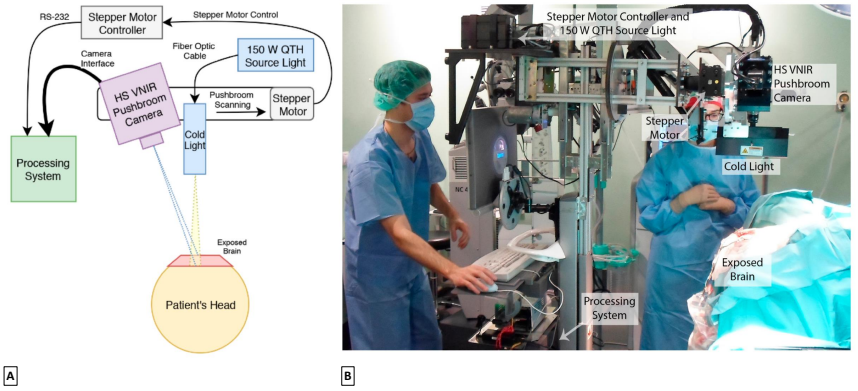


Figure 3.3: Hyperspectral acquisition system. A) Schematic diagram of the acquisition system. B) The acquisition system capturing an hyperspectral image during a neurosurgery at the University Hospital Doctor Negrin of Las Palmas de Gran Canaria (Spain) [63].

images. During the surgery, after the craniotomy and the dura resection, the operating surgeon places some rubber ring markers in some points that he has already identified as healthy or cancerous tissue. The identification is based on information provided by the neuronavigator from an MRI scan (Magnetic Resonance Image). The image acquisition takes place while the markers are in situ. The scenes can be captured in different stages during a surgery on the base of the tumor location: if it is a superficial cancer the HS cube can be captured immediately after the dura resection. If the tumor is in a deeper position, the image is acquired during the actual tumor resection. Once the camera has acquired the image, some tissue samples inside the markers placed in the tumor area are resected and sent to the pathologists to obtain the specific histopathological diagnosis (grade and type of tumor). Markers on healthy tissue are placed only as reference for the labelling process since it is not ethic to remove normal tissue for the biopsy causing damages to the patient. The histopathological information provided by the biopsy and the surgeons knowledge are used to label the acquired HS cubes and generate a gold standard dataset for the supervised classification phase of the brain cancer detection system [48]. In order to label the whole HS cube it has been used a tool designed in Matlab and based on the Spectral Angle Mapper (SAM). The SAM classification compares the spectra of the image pixels with a well-known spectrum obtained from a reference pixel [48]. After the medical operation, the surgeon uses this tool to generate the gold standard map related to the images. The aim is to assign to each pixel one of these: *healty tissue*, *tumor tissue*, *hypervascularized tissue* and *background*. The HS cube taken during the surgery is the input of the tool: the user can watch the correspondent RGB image and select the reference pixel in the position where the biopsy has been done (within the ring marker for the tumor class) or far enough from the cancer where the surgeon is confident that the area corresponds to the other classes. On the base of the SAM measurement, the tool shows the most similar pixels to the selected reference one. The user can set a threshold to vary the tolerance which is at the base of the pixel selection. Once the user considers that all the pixels displayed belong to a single class, those pixels are labelled with that class value. In Figure 3.4 it can be noticed how this

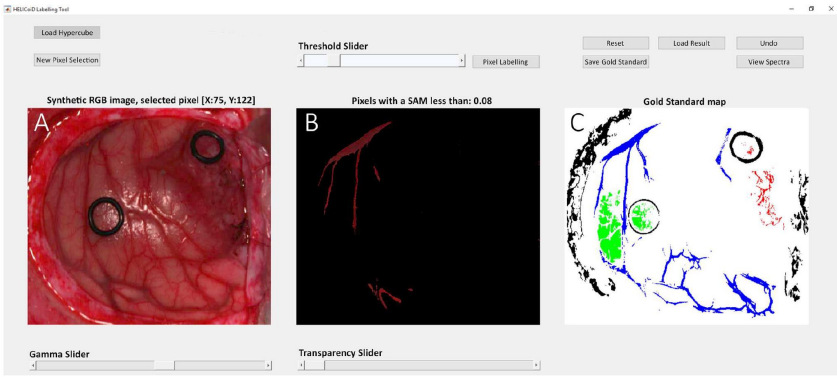


Figure 3.4: HELICoiD Labelling Tool A) RGB imge of the HS cube; B) Pixels that belong to the hypervascularized class; C) Gold standard map of the HS cube [48].

tool works. On the left (Figure 3.4.A), there is the RGB image related to the HS cube under study. In the middle (Figure 3.4.B), the pixels with a spectral angle whose value is less than 0.08° with respect to the reference pixel (that, in this case, belongs to the blood vessel -hypervascularized tissue class) are displayed. On the right, Figure 3.4.C shows the gold standard map of the HS cube where the green color refers to the healthy tissue, the red to the tumor, the blue to the hypervascularized tissue and the black to the background. A total of 44,555 spectral signatures have been labelled exploiting this tool. Table 3.1 provides information about the five images which belong to the dataset used in this work.

3.4 Brain cancer detection system

The system developed in the HELICoiD project performs a hybrid classification of the hyperspectral image acquired during the surgery. Figure 3.5 shows an overview of the system [48]. At first, the image is stored as a *raw* file that is pre-processed in order to homogenize the spectral signature of each pixel. The pre-processed image is the input of the two main

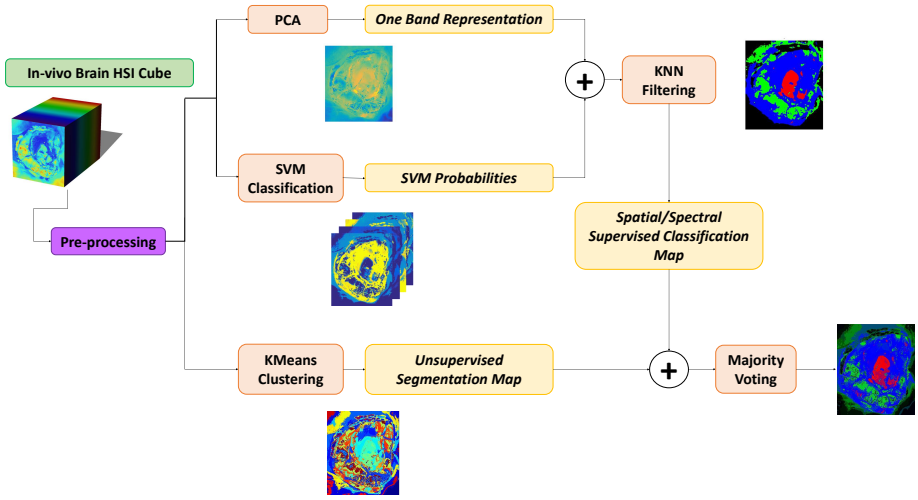


Figure 3.5: Algorithms present in the brain cancer detection system. The HSI cube is the input of the system. It is pre-processed and sent to the supervised (Principal Component Analysis, Support Vector Machine and K-Nearest Neighbors) and unsupervised (K-means) classification branches of the system. The results of these two flows are combined with the Majority Voting algorithm. The output is the classification maps which displays the classes with different colors.

parts of the system: one performs a spatial-spectral supervised classification while the other accomplishes an unsupervised clustering segmentation.

Image ID	#Pixel	Dimensions (Width × Height × Bands)
P1C1	251,532	548×459×826
P1C2	264,408	552×479×826
P2C1	219,232	496×442×826
P3C1	185,368	493×376×826
P4C1	124,691	329×379×826

Table 3.1: HS brain cancer image database. The table presents the image ID, the number of pixels and the initial number of bands [63].

Concerning the supervised classification, the Principal Component Analysis (PCA) computes a one-band representation of the HS cube, while the Support Vector Machine (SVM) performs a pixel-wise classification and generates a probability map. The K-Nearest Neighbors (KNN) algorithm is used as a filter to integrate the PCA and the SVM outputs in order to improve the results of the spectral classification by adding spatial domain information [63]. As for the unsupervised stage of the system, it is based on the K-means algorithm which creates a segmentation map where the boundaries of the different tissues displayed in the image are well delineated. The Majority Voting algorithm combines these two main parts of the system whose output is the result of a hybrid classification. In the following paragraphs, a more detailed description of the algorithms shown in Figure 3.5 and of their implementations are introduced. In particular, more emphasis will be put in the PCA, SVM and KNN algorithms since their serial and parallel implementations were developed in this work, while the Pre-processing and K-means versions have been developed in other thesis. Concerning the Majority Voting (MV), the new system developed in this work will contain the same version already present in the original system. It is important to highlight that the original serial version of the brain cancer detection system had some parts written in C language and other which exploits the OpenCV library [65]. The first aim of this work is to develop a version of the complete system in C language in order to have an optimized serial version which will also be the basis for the parallel implementation written both in C and in CUDA language to exploit the GPU technology. For this reason, the first step of the work is the serial version development of each algorithm. Figure 3.6 shows the flow of the serial version of the complete system. It is worth noting that all the algorithms are executed in a sequential way, even if the spatial spectral classification part of the system (PCA+SVM+KNN) could be executed simultaneously with the K-means part (Figure 3.5), since they are independent. Once the serial version is completed, an analysis of the performances of the algorithm steps is performed. This evaluation allows to understand which are the heaviest parts from the computational point of view and if these steps can be executed in parallel. At this point, one or more parallel versions of each algorithm are

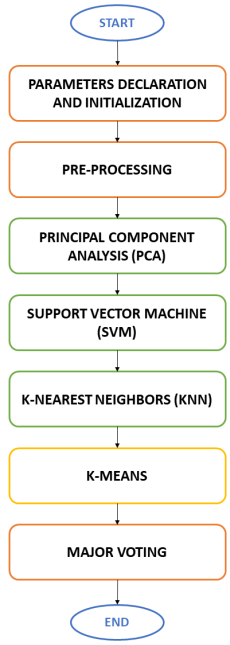


Figure 3.6: Serial flow of the brain cancer detection system. The image shows the sequential flow of the serial code.

developed. Both the serial and parallel results of each algorithm are compared in order to select the best version which will be part of the complete system. The results shown in the following paragraphs are obtained using *System 1* and *System 2* already described in Paragraph 1.2.2.

3.4.1 Pre-Processing

The algorithm and the serial version

The captured image could present significant signal variations due to a non-uniform illumination of the brain surface. The pre-processing al-

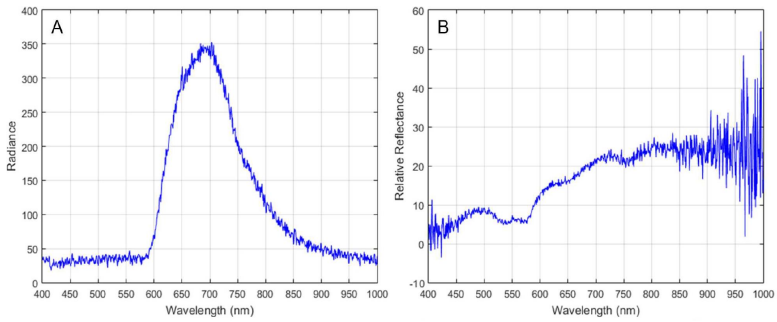


Figure 3.7: Raw and calibrated spectral signatures of the grade IV glioblastoma tumor tissue. A) Raw spectral signature of a grade IV glioblastoma tumour tissue; B) Result of the image calibration. [48]

gorithm aims to correct these differences and to homogenize the spectral signatures of the labelled samples obtained from the in-vivo hyperspectral data-cubes [62]. The pre-processing chain is made up of three main steps: the image calibration, the band reduction and the data normalization. To correct the signal variations of the input data, in the calibration phase a *white* and *dark* images are acquired in the same operation theatre and with the same illumination conditions where the HS cube will be captured. The white image is obtained using a white standard reference tile while the black one keeping the shutter camera closed. The calibrated image (CI in Equation 3.1) is computed as follows:

$$CI = 100\frac{RI - DR}{WR - DR} \tag{3.1}$$

where RI is the input image, WI and DI are the white and dark images [62]. Figure 3.7 shows the raw (A) and the calibrated (B) spectral signature of the grade IV glioblastoma tumor tissue. Once the image is calibrated a dimension reduction is performed. The aim is not to consider those bands that contain high noise. In fact, in the bands ranges $0 \div 55$ and $750 \div 826$ the

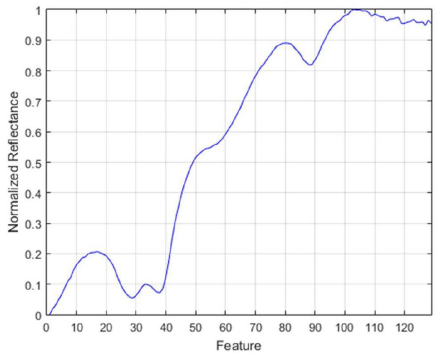


Figure 3.8: Normalized spectral signature of the grade IV glioblastoma tumor tissue [48].

camera sensor is underperforming thus these bands do not give useful information. Furthermore, it has been observed that consecutive bands provide redundant information and this is due to the extremely high spectral resolution [48] [62]. For this reason, the reflectance value of seven contiguous bands are averaged every five bands: the reduced HS cube is characterized by 128 bands [61]. This reduction allows to save computational time and memory due to the limited number of data to process. The last phase of the pre-processing chain is the data normalization which is needed to avoid high radiation intensity differences between the pixels captured at different heights. This fact could produce a classification based on the brightness of the pixel without really taking into account its spectral signature. For this reason a normalization of the pixels brightness is needed without modifying the spectral signature. Figure 3.8 shows the result of this phase. Finally, a further noise filtering could be introduced in the system in the second phase. It is the first stage of the hyperspectral signal identification by minimum error (HySime) algorithm called *Hyperspectral Noise Estimation* [48]. In this work this noise filtering is not considered but it is an option that the user can include in the final system configuration, as it will be explained in Paragraph 3.7.

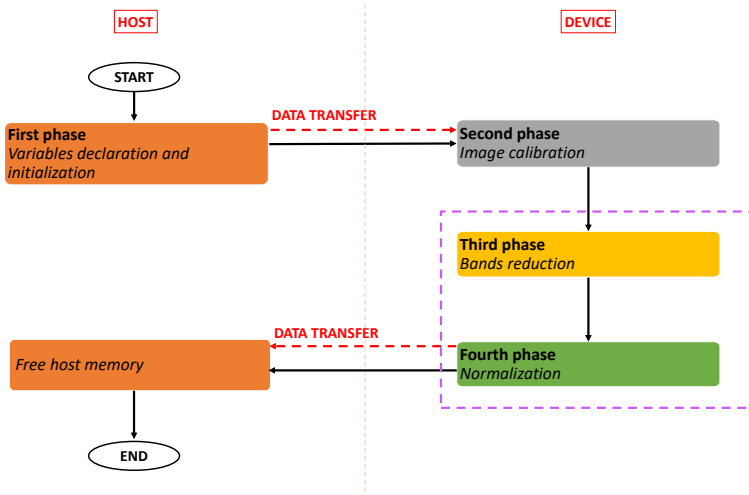


Figure 3.9: Flow of the Pre-processing parallel code.

The parallel version

The parallel version of the pre-processing algorithm has been developed in the master thesis work cited above. The flow of the parallel code (Figure 3.9) starts on the host where the input image and the white and dark images are read and stored. These data are then transferred to the device where the pre-processing chain can start with the calibration (*Second phase*). It is important to highlight that the data transfer from host to device is managed using the CUDA *streams* [23] which allow to overlap the data transfers and the operations performed on the device. In this case, a number of streams equal to the rows of the matrices is created. Each stream manages the transfer of a row and the computation of the calibrated image is performed exploiting the cuBLAS library (NVIDIA) which includes highly optimized linear algebra routines [66]. Once the row transfer is completed, the cuBLAS function can start the computation without waiting the transfer of the following row. In this way, the computational time is reduced compared to a serial elaboration. Once the image is calibrated, the following two phases can start. In particular, a custom kernel performs the

normalization while the band reduction is done sending to this kernel only the final bands that have to be considered. In this way, the *Third* and the *Fourth Phase* are performed together. At the end, the pre-processed image is transferred to the host.

3.4.2 Principal Component Analysis

The algorithm and the serial version

Despite being characterized by a large amount of data, even after the reduction obtained with the previous step, the HS image dimensionality can be decreased even more through the Principal Component Analysis (PCA). This technique converts the original data into a subspace with smaller dimensions, and selects and accumulates the most important information in the first bands [67]. PCA reduces those data computing the covariance matrix of the HS cube and its eigenvector decomposition. Then the image is projected into the sub-space described by these eigenvectors and, at the end, the principal components or bands are selected. The serial implementation of the PCA algorithm is organized in six steps (Figure 3.10) [67]: after the variables declaration and initialization (*First phase*), the algorithm computes the transpose of the input matrix Y_{NxM}, where N is the number of pixels and M is the number of bands (*Part A*). The second step (*Part B*) continues to compute the average of the elements present in each row of Y^T. Then, each average is removed from each element of the corresponding band (i.e. the row of the matrix Y^T) (*Second phase*). The covariance matrix C_{MxM} is then computed as the product between the matrix X_{MxN} (*Third phase*), output of the second step, and its transpose (Equation 3.2).

$$C_{MxM} = X^T \cdot X \tag{3.2}$$

In the next step the eigenvectors E, associated to the covariance matrix C, are extracted exploiting the Jacobi method (*Fourth phase*). This method has been selected because it is a fast algorithm capable of extracting the eigenvectors while computing the input matrix eigenvalues [67]. The fifth step (*Fifth phase*) performs the projection of the image Y onto the set of

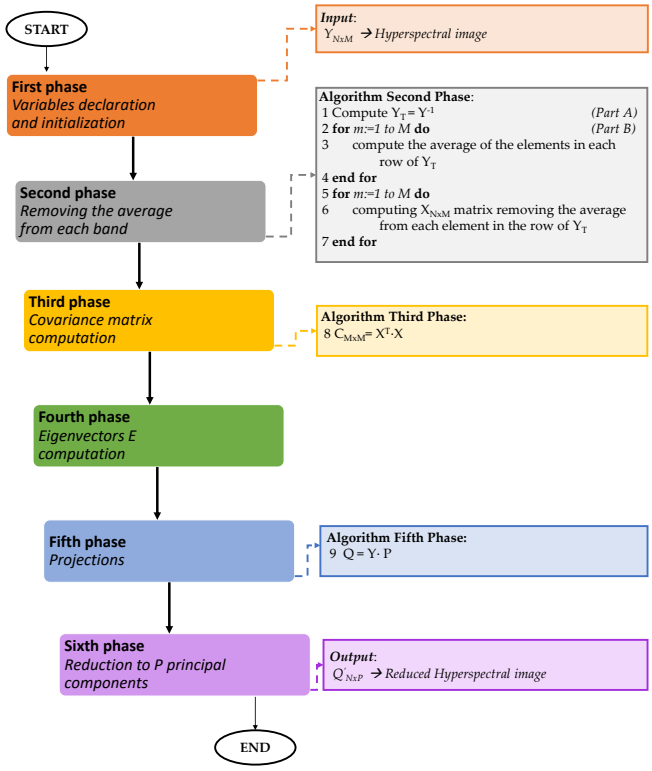


Figure 3.10: Steps of the PCA algorithm.

the extracted eigenvectors (Equation 3.3).

$$Q = Y \cdot E \tag{3.3}$$

Finally, the first P principal components are selected in order to obtain a matrix with reduced dimension $N \times P$ (*Sixth phase*). In this work, the output consists in an array whose dimension is $N \times 1$, since the number of principal components selected is one.

The parallel versions

As said in the previous paragraph, the goal of the PCA algorithm is to further reduce the number of bands of the pre-processed HS cube, which represents the input. In this case, the output is an array since the number of principal component is one. The first analysis conducted aimed to understand which steps were heavy from the computational point of view. Then, they were studied in order to understand which was the more efficient technique to develop their parallel version. Considering the steps presented in Figure 3.10, the most consuming part is the covariance matrix computation that takes more than the 70% of the total time of the serial execution. In the development of the first parallel version (P1), two options have been evaluated: the former executes only the *Third phase* on the GPU and leaves all the other steps on the host, since they are faster than the covariance matrix computation. The latter parallelizes also the other phases trying to reduce their computational time since the conducted analysis showed that also those operations can be parallelized. Another aspect that has to be taken into account is the number of data transfers that, in these two cases, is the same. If only the *Third phase* has to be performed on the GPU there will be a memory transfer between the *Second phase* and the *Third phase* (from host to device) and another transfer after the covariance matrix computation (from device to host). If all the steps are computed on the device there will be an initial memory transfer (from host to device) at the beginning of the computation and another one (from device to host) at the end. This second solution is characterized by data transfers with lower sizes than the first option: in fact, in both cases the transfer from host to device has a size equals to $M \times N \times dim_{float}$, where M is the number of bands, N the number of pixels and dim_{float} is the size of a *float*, i.e. 4 bytes. The transfer from device to host has different dimensions in the two options: in the first case is $M \times M \times dim_{float}$ while, in the second case, is $N \times P \times dim_{float}$, where P is the number of principal components, that in this work is set equal to 1. For these reasons, the second option has been selected for the first parallel version P1. The flow of the first parallel version is shown in Figure 3.11. The code starts from the host where the

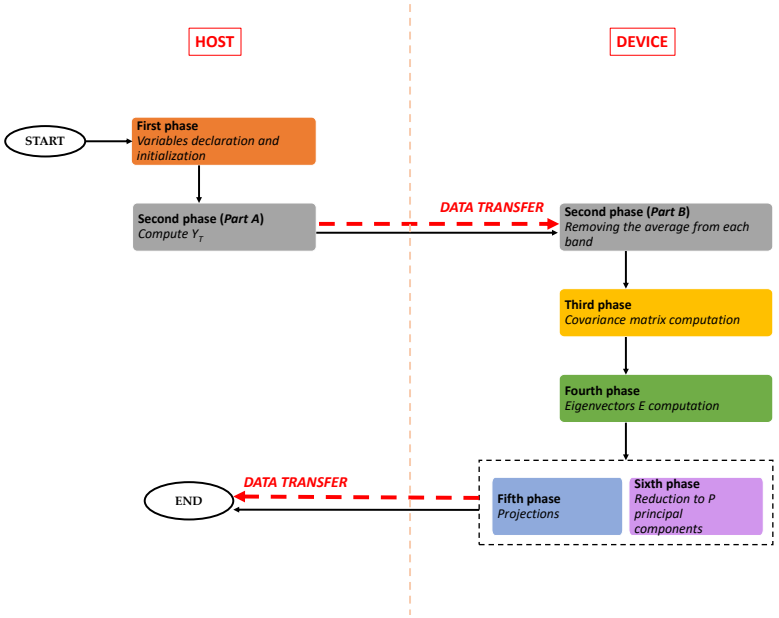


Figure 3.11: First version of the PCA parallel code.

variables declaration and initialization are performed (*First phase*). As it can be seen in the figure, the *Second phase* is divided in two parts: *Part A* is sequentially performed by the CPU and concerns the computation of the transpose of the input matrix (Y_T). After the GPU memory initialization, this new matrix is transferred from the host to the device memory where *Part B* starts. In this step two main operations are performed: the average computation of each row of the matrix Y^T and the subtraction of these averages from each elements of the corresponding rows. The first task is evaluated exploiting the highly optimized function *cublasDasum* that belongs to the cuBLAS library [66]. It must be noticed that the data transfer from host to device is managed using *streams*. In this case, a number of streams equal to the number of bands (i.e. the number of rows of the matrix) is created. Each stream manages the row transfer and the average computation of its elements through the *cublasDasum* function. Once the

transfer is completed, the *cublasDasum* function can start without waiting for the next transfer. In this way, the computational time is saved. After that all the averages are computed, they are subtracted from each element of the related row. This computation is performed by a custom kernel. Also in this case, each *stream* manages the call of the kernel which removes the average from each element simultaneously. This kernel is characterized by a *grid* whose dimension is computed according to Equation 3.4:

$$\dim_{grid} = \frac{N}{n_{threads}} \tag{3.4}$$

where N is the number of pixels present in the image and $n_{threads}$ is the number of threads that are in one block of the grid. In this case, this number is set to 32 according to the definition of *warp* provided by NVIDIA. If the remainder of Equation 3.4 is not equal to zero, dim_{grid} is incremented by one. The output of the *Second phase* is the matrix X_{MxN} where M is the number of bands and N the number of pixels. X represents the *Third phase* input where the covariance matrix C is computed as described in Equation 3.2. This step exploits another function of the cuBLAS library, *cublasDgemm*, which allows to compute the product between two matrices. The end of the *Third phase* is characterized by another custom kernel which simultaneously divides each element of the covariance matrix C by the number of pixels. The *Fourth phase* extracts the eigenvectors of the covariance matrix which is the output of the previous step. As previoulsy said, the Jacobi method has been selected for the eigenvectors computation. A suitable function of the cuSOLVER library [68] (developed by NVIDIA) has been employed to compute the eigenvalues and the eigenvectors in order to exploit the intrinsic parallelism of the Jacobi method. In order to use this routine, called *syevj*, some variables and arrays have to be declared and initialized. These variables, together with the covariance matrix, are the inputs of the function which returns the sorted eigenvectors matrix. In the case of the parallel code, the *Fifth phase* and the *Sixth phase* are performed together: the projection of the original image into the eigenvectors is performed by a cuBLAS function and the inputs of this routine are only the eigenvectors related to the principal components that have to be

selected. It is important to highlight that if the number of principal components to select is equal to one, the projection (already described in Equation 3.3) is a product between a matrix (the original image) and a vector (the eigenvectors). In this case the chosen cuBLAS function is *cublasDgemv*. If the number of principal components to select is higher than one, the projection is the result of a matrix-matrix product and it is performed by a *cublasDgemm*. In this way, the result of this computation has already been reduced because its dimension is equal to $N \times P$, where N is the number of pixels and P the number of principal components selected. At the end, the result is transferred to the host. Analyzing the performance of this parallel version, it is possible to notice that the *Fourth phase* takes longer than the corresponding step in the serial code. For this reason, another parallel version of the code has been developed in order to evaluate if moving the eigenvectors computation on the host is more efficient even with the incremented number of needed transfers. The flow of the second parallel version of the PCA is shown in Figure 3.12. In this new version, the covariance matrix is transferred from the device to the host after its computation in the *Third phase*. On the host side, the Jacobi method is performed to find the eigenvectors of the covariance matrix (*Fourth phase*). The principal components P selection, which constituted the *Sixth phase* of the previous version, is moved to the host. In fact, only the eigenvectors related to these components are copied from the host to the device in order to reduce the transfer time and the memory occupancy. At this point the projections are computed as described in the previous parallel version, exploiting the cuBLAS functions. Finally, the PCA result is transferred from the device to the host.

PCA results

In this paragraph, the results of the executions of the serial and parallel PCA codes are presented. Table 3.2 shows the processing times related to the serial and the two parallel versions. In particular, the times related to the single phases of the algorithm are shown in order to understand which are the heaviest parts of the algorithm and how their execution time

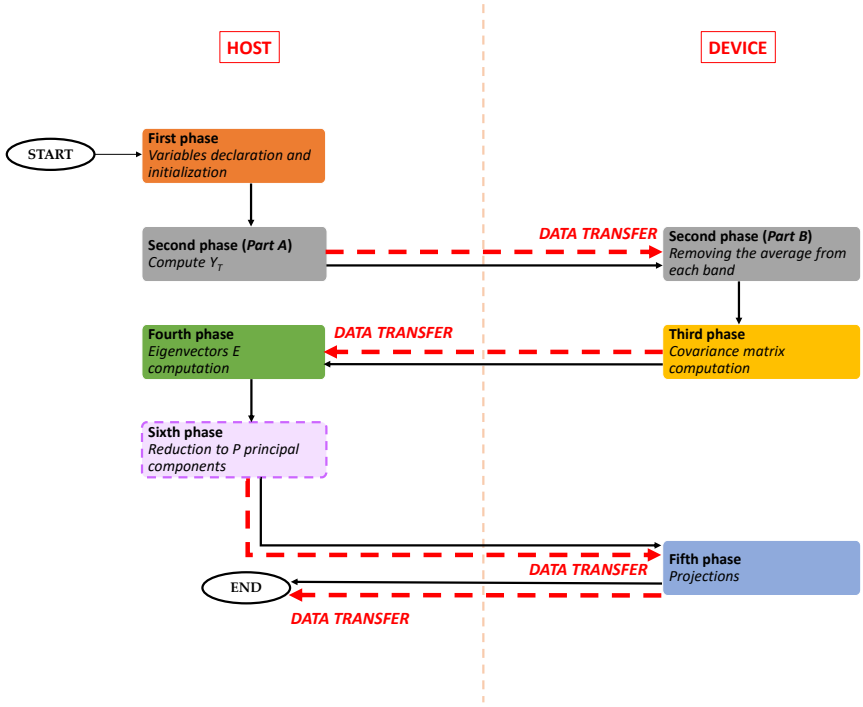


Figure 3.12: Second version of the PCA parallel code.

changes in the different versions. These tests have been carried out on the *System 1* presented above exploiting one of the two boards for the parallel code executions. Concerning the elaboration times of the serial versions, it is possible to notice that the covariance matrix computation is the heaviest part of the execution (*Third phase*). The percentage of time dedicated to this step is higher than 74% in all the executions and it reaches the 82.38% if the P4C1 image is considered. For this reason, this step is the one that has to be parallelized. It can be also noticed that both in the serial and in the parallel execution, the time is proportional to the pixels number. In fact, as this number increases, the computations and the memory transfers (in the parallel codes) are heavier. As said in the previous paragraph, the strategy adopted in the developing of the first parallel version (P1 in Table

Image ID	Version	Time *First phase* and *Second phase - Part A*	Time *Second phase - Part B*	Time *Third phase*	Time *Fourth phase*	Time *Fifth phase*	Time *Sixt phase*	Total Time
P1C1	S	0.29 (12.94%)	0.07 (2.91%)	1.83 (80.75%)	0.01 (0.26%)	0.07 (3.13%)	0	2.271
	P1	0.58 (56.12%)	0.05 (4.63%)	0.02 (2.31%)	0.13 (12.82%)	0.001 (0.096%)	0	1.037
	P2	0.56 (58.41%)	0.05 (5.02%)	0.02 (2.51%)	0.01 (0.73%)	0.002 (0.21%)	0	0.957
P1C2	S	0.32 (12.15%)	0.07 (2.58%)	1.95 (74.03%)	0.01 (0.19%)	0.29 (11.05%)	0	2.634
	P1	0.50 (46.48%)	0.05 (4.63%)	0.03 (2.31%)	0.13 (12.31%)	0.001 (0.09%)	0	1.08
	P2	0.58 (59.67%)	0.05 (5.02%)	0.03 (2.56%)	0.01 (0.51%)	0.002 (0.20%)	0	0.977
P2C1	S	0.25 (12.61%)	0.06 (2.83%)	1.60 (80.88%)	0.003 (0.15%)	0.06 (3.13%)	0	1.982
	P1	0.51 (51.67%)	0.04 (4.16%)	0.02 (2.03%)	0.13 (13.50%)	0.001 (0.10%)	0	0.985
	P2	0.45 (57.67%)	0.04 (5.24%)	0.02 (2.56%)	0.004 (0.51%)	0.001 (0.13%)	0	0.782
P3C1	S	0.21 (12.37%)	0.05 (2.88%)	1.35 (80.96%)	0.003 (0.18%)	0.05 (3.00%)	0	1.665
	P1	0.45 (50.22%)	0.04 (3.87%)	0.02 (1.88%)	0.13 (14.71%)	0.001 (0.11%)	0	0.904
	P2	0.39 (55.35%)	0.04 (4.93%)	0.02 (2.39%)	0.004 (0.56%)	0.002 (0.28%)	0	0.71
P4C1	S	0.13 (11.44%)	0.03 (2.91%)	0.91 (82.38%)	0.003 (0.27%)	0.03 (2.81%)	0	1.101
	P1	0.37 (51.74%)	0.02 (3.35%)	0.01 (1.67%)	0.13 (17.71%)	0.001 (0.14%)	0	0.717
	P2	0.34 (54.20%)	0.03 (4.04%)	0.01 (1.94%)	0.004 (0.65%)	0.001 (0.16%)	0	0.618

Table 3.2: PCA results. The table shows the computational times related to the serial (S), the first parallel (P1) and the second parallel (P2) versions execution exploiting the *System 1*. Concerning the parallel code, only one GPU board present in the system has been used. For each version, the times of the single phases and of the entire execution are presented. The times are the average of five executions and are expressed in seconds [s].

3.2) is to perform all the steps, except the first one, on the device as already presented in Figure 3.11. Analyzing the results of the P1 versions, it is possible to notice that the time of the *Third phase* is considerably reduced since it reaches 0.02 s and 0.03 s in the cases of the P1C1 and P1C2 images, which corresponds to about 2.30% of the entire computational time. Concerning the other steps, their times are lower than the serial version with two exceptions: the *First phase-Second phase - Part A* and the *Fourth phase*. The former, performed on host (see Figure 3.11), contains all the variables declaration and initialization, a suitable data packaging for an efficient data transfer and the *Part A* of the *Second Phase* is the input matrix transposition. This last step is left on the host since it is equally well-performing. The *Fourth phase* concerns the eigenvector computation, which is performed by a cuSOLVER function, as explained before. In this case, over the 90% of this phase time is related to the variables declaration and initialization used to exploit the cuSOLVER function (which takes the

Image ID	Version	Speed-up
P1C1	P1	2.19
	P2	2.37
P1C2	P1	2.44
	P2	2.70
P2C1	P1	2.01
	P2	2.53
P3C1	P1	1.84
	P2	2.35
P4C1	P1	1.54
	P2	1.78

Table 3.3: PCA speed-up (*System 1*). The two parallel version P1 and P2 are compared with the serial version S and the speed-up are computed as the ratio between the serial and the parallel computational time.

$\sim$5% of the time to compute the eigenvectors). As said in the previous paragraph, the second parallel version (P2) has been developed in order to decrease the time of the *Fourth phase*: the idea was to perform the eigenvector computation on the host and to continue the execution on the device. Even if the number of transfers in P2 is higher than in P1, the total time of the computation is lower because there is a substantial difference in the times related to the two different eigenvector computations. It is possible to highlight that the time of all the other steps does not present significant differences from P1 and P2. Furthermore, it is possible to notice that the *Fourth phase* time in S and P2 is almost the same since in both cases this step is performed on the host. The faster times of the P2 version highlights that, in this case, the eigenvectors computation with the cuSOLVER function is heavier than a higher number of data transfers. Considering the P2 version, the biggest database image (P1C2 with 264,408 pixels) is processed in less than one second (0.977 s) compared to the 2.634 s of the serial version. Table 3.3 presents the speed-up of the two parallel versions P1 and P2 compared to the serial one S. These speed-up show that the P2 version obtains the higher values considering all the images of the database.

Image ID	Version	Time *First phase* and *Second phase - Part A*	Time *Second phase - Part B*	Time *Third phase*	Time *Fourth phase*	Time *Fifth phase*	Time *Sixt phase*	Total Time
P1C1	S	0.31 (10.77%)	0.07 (2.58%)	2.35 (82.80%)	0.003 (0.11%)	0.05 (1.91%)	0	2.832
	P1	0.73 (51.7%)	0.03 (2.19%)	0.08 (5.60%)	0.28 (20.04%)	0.002 (0.14%)	0	1.412
	P2	0.66 (57.77%)	0.03 (2.62%)	0.08 (6.89%)	0.01 (0.44%)	0.002 (0.18%)	0	1.146
P1C2	S	0.32 (10.56%)	0.08 (2.78%)	2.45 (82.17%)	0.003 (0.10%)	0.073 (2.48%)	0	2.983
	P1	0.73 (50.45%)	0.03 (2.20%)	0.08 (10.77%)	0.22 (15.21%)	0.002 (0.14%)	0	1.453
	P2	0.64 (51.16%)	0.03 (2.39%)	0.08 (6.31%)	0.004 (0.32%)	0.002 (0.16%)	0	1.253
P2C1	S	0.28 (11.02%)	0.07 (2.68%)	2.05 (84.43%)	0.002 (0.08%)	0.06 (2.48%)	0	2.422
	P1	0.68 (47.32%)	0.03 (1.96%)	0.07 (5.21%)	0.24 (17.74%)	0.002 (0.15%)	0	1.325
	P2	0.55 (52.08%)	0.03 (2.46%)	0.07 (6.52%)	0.003 (0.28%)	0.002 (0.19%)	0	1.058
P3C1	S	0.25 (12.21%)	0.05 (2.56%)	1.68 (82.77%)	0.002 (0.09%)	0.42 (20.68%)	0	2.031
	P1	0.51 (38.16%)	0.02 (1.48%)	0.06 (4.31%)	0.23 (17.08%)	0.001 (0.07%)	0	1.347
	P2	0.53 (54.33%)	0.02 (2.35%)	0.06 (5.91%)	0.002 (0.20%)	0.002 (0.20%)	0	0.981
P4C1	S	0.15 (10.64%)	0.04 (2.55%)	1.12 (81.71%)	0.003 (0.22%)	0.03 (1.97%)	0	1.372
	P1	0.43 (38.50%)	0.02 (1.34%)	0.04 (3.48%)	0.20 (18.00%)	0.001 (0.09%)	0	1.122
	P2	0.43 (50.35%)	0.02 (1.761%)	0.04 (4.58%)	0.004 (0.47%)	0.001 (0.12%)	0	0.852

Table 3.4: PCA results. The table shows the computational times related to the serial (S), the first parallel (P1) and the second parallel (P2) versions execution exploiting the *System 2*. For each version, the times of the single phases and of the entire execution are presented. The times are the average of five executions and are expressed in seconds [s].

These three versions are also executed exploiting the *System 2*. The previous analysis can be also done in this case and the same conclusions can be drawn. The serial results are slightly higher than the previous ones but it is possible to notice that the covariance matrix computation *Third phase* is still the heaviest execution part. Concerning the P1 and P2 versions, the *First phase and Second phase-Part A* is slower than the respective value of the serial execution: this step is performed on the host in all the versions but, in the parallel versions, it contains also the data preparation for the transfer. The speed-up obtained with the parallel versions are shown in Table 3.5. From Table 3.5 stands out that even in this case the P2 version is the fastest one. Compared with the P2 results obtained with *System 1* (Table 3.2), the ones obtained with *System 2* are slower. In fact, it is important to consider that the Tesla K40 is equipped with a number of cores that is almost twice the GTX1060 one. Furthermore, the Tesla K40 is connected to the CPU through a PCI Express 3.0, while the GTX1060 is connected

Image ID	Version	speed-up
P1C1	P1	2.01
	P2	2.47
P1C2	P1	2.05
	P2	2.38
P2C1	P1	1.83
	P2	2.29
P3C1	P1	1.51
	P2	2.07
P4C1	P1	1.22
	P2	1.61

Table 3.5: PCA speed-up (*System 2*). These speed-ups are obtained using the second system. These values are the ratio of the serial and parallel computational times.

through a PCI Express 2.0: this means that in the first system the data transfers are faster. Finally, exploiting the NVIDIA Visual Profiler it is possible to evaluate the percentage of the computational time dedicated to the data transfer that, in this case, is equal to the 0.90%. This value highlights that the data preparation performed in *First phase* is efficient. To graphically analyze the PCA results, Figure 3.13 shows the one-band representation of the image P2C1 when the PCA algorithm is applied. It is possible to notice the spatial representation of the most relevant spectral information after the data dimensionality reduction.

3.4.3 Support Vector Machine

The algorithm and the serial version

The Support Vector Machine (SVM) is a machine learning technique which computes the probability of a pixel to belong to each class under study. The highest probability corresponds to the class assigned to the pixel. In this system, the output of the SVM classifier is not the label that represents the class but a probability map which contains the probabilities of each pixel to belong to the four classes. This output will be one of the

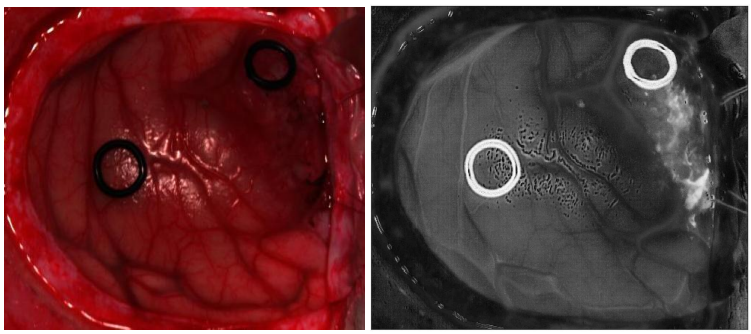


Figure 3.13: PCA result. The image on the left (*Image ID=P2C1*) is the input of the algorithm, so the HS cube acquired during the neurosurgical operation. The image on the right is the one band representation of the input.

inputs of the KNN algorithm, which is the following step. This classifier was selected for the brain cancer detection system because it provides good performance for this kind of data and in the present experiment conditions: limited training dataset and real-time constraint [48] [69]. The SVM algorithm is composed of two main parts: training and prediction. Since the SVM performs a supervised classification, a labelled dataset is needed in order to train the model which will be used in the prediction phase. This dataset is developed following the procedure explained in Paragraph 3.3: the neurosurgeons and the pathologists, on the base of their knowledge, experience, of the biopsy results and of the output of the SAM algorithm, assign a label to some of the pixels. Since it is impossible to know the spectral signatures, the labels, of each HS cube pixel, there are two ways to evaluate the performance of the generated model. Concerning the labelled pixels, it is possible to use standard metrics, as *sensitivity* and *specificity*, to evaluate the accuracy of the model when it has to predict new data. Regarding the whole HS cube, not entirely labelled, the performance of the supervised model can only be estimated through a visual evaluation of expert neurosurgeons [48]. The training phase of the SVM classifier is developed using MATLAB, exploiting the LIBSVM library. It is not present

in this thesis since it is considered an *offline* step of the system: the idea is to generate the supervised model and, then, to send it as an input to the prediction part of the algorithm. The SVM algorithm faces a multiclass problem with a *one-against-one* or binary strategy: each pair of classes are evaluated in order to find the best hyperplane that separates their elements in the space. The number of binary evaluations, and so the number of the computed linear hyperplanes, depends on the number of classes considered in the classification, as shown in Equation 3.5.

$$n_{hp} = \frac{n_{class} * (n_{class} - 1)}{2} \tag{3.5}$$

In this work the number of hyperplanes n_{hp} is six since the number of classes n_{class} is four. Each hyperplane generated in the training phase is associated to a w vector which is computed as the sum of the support vectors, each of them modified by the Lagrange multiplier and the class value. The w vector is described by Equation 3.6:

$$w = \sum_{i \in S} \alpha_{ij} \cdot v_j \cdot x_i \tag{3.6}$$

where x_i represents a set of support vectors selected among the training set S as the closest samples of their class to the binary hyperplane, y_i is the class of each support vector and α_i is the Lagrange multiplier associated to a sample [69]. The linear discriminant function *f(x)* associated to each binary classifier is shown in Equation 3.7:

$$f(x) = w \cdot x + b \tag{3.7}$$

where x is the sample that has to be classified and b is the bias of the hyperplane that separates the two classes. The serial SVM classification algorithm [69] is characterized by three main phases: the variables declaration and initialization, the binary probability computation and the multiclass probability computation (Figure 3.14). The SVM prediction algorithm inputs are: the sample to classify (x), the matrix W which contains all the *w-vectors* (w_{ij}), B the matrix of biasses (b_{ij}), Sa and Sb which are parameters of the sigmoid function and the HS cube. The indexes i and j refers

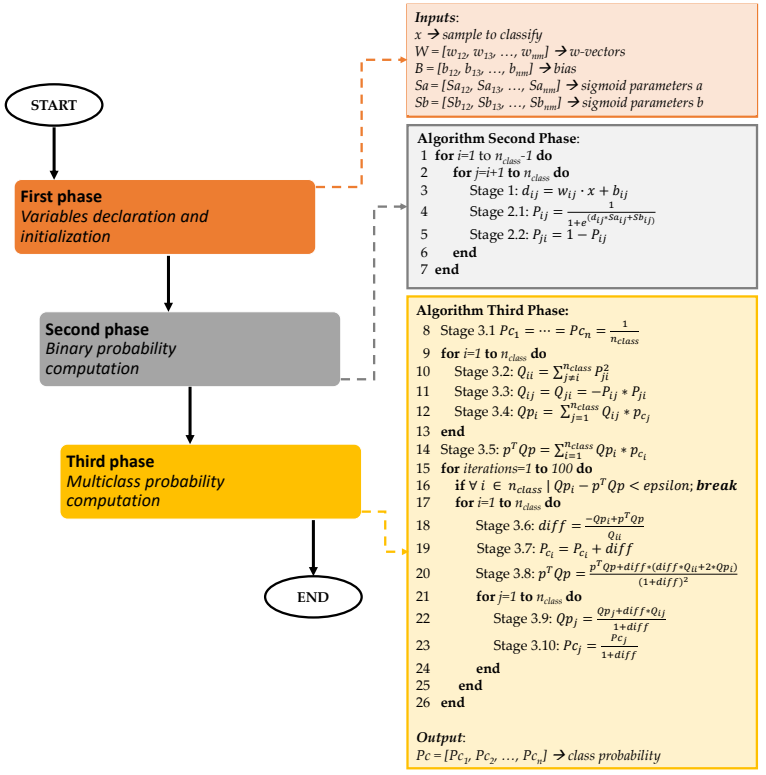


Figure 3.14: SVM serial algorithm.

to the classes of each binary classifier [69]. In Figure 3.14, the *Stage 1* of the algorithm (*line 3*) computes the distance (d_{ij}) of the sample x from the hyperplane applying the discriminant function described in Equation 3.7. The second stage (*lines 4-5*) estimates the probabilities of the sample to belong to the two classes under study. In particular, *Stage 2.1* computes the probability of the sample, associated to one of the classes, through a sigmoid function to the relative distance. *Stage 2.2* computes the probabil-

ity of the sample to belong to the other class of the classifier. These first steps are executed a number of times equal to the number of hyperplanes. *Stage 3* computes the multiclass probabilities combining the binary probabilities generated in the previous stages. At first, the class probabilities are initialized with the same value (*line 8*). *Stage 3.2* and *Stage 3.3* (*lines 10-11*) compute the matrix Q which is obtained from the binary probabilities calculated before. The steps from *Stage 3.4* to *Stage 3.10* (*lines 12 – 23*) aim to compute the multiclass probabilities: the *for* loop (*lines 15 – 26*) iteratively refines the values of the probabilities of a pixel associated to a class. The value of each probability is incrementally modified as long as the difference with the value of the previous iteration is under a threshold or if the maximum error (*epsilon*) is reached. When one of these two cases is verified, the multiclass probabilities of the sample are computed. Both the threshold and the error are set by the user and, for this reason, they are two algorithm inputs. The SVM output consists of a probability map containing the multiclass probabilities of all the samples, that are the pixels of the HS image. It is important to highlight that all the steps shown in Figure 3.14 are computed independently for each sample: this will be an essential aspect in the develop a pixel-wise classification.

The parallel version

Even if the analysis conducted on the serial SVM results (presented in the following paragraph) proves that it is a very optimized algorithm since its computational time is very low, a parallel version of this algorithm has been developed anyway. The reason is that a parallel version of this classifier could be very useful in the development of the complete system by avoiding some data transfers. In fact, if the complete system will perform the PCA and the KNN on the device, performing even the SVM on GPU could be more efficient thanks to the fewer I/O transfers from the host to the device and viceversa. The fact that the multiclass probability computation is independent for each sample eases the development of the parallel version of the SVM algorithm. In Figure 3.15 the main steps of the code are presented. The flow starts on the host where the inputs reading and the variables

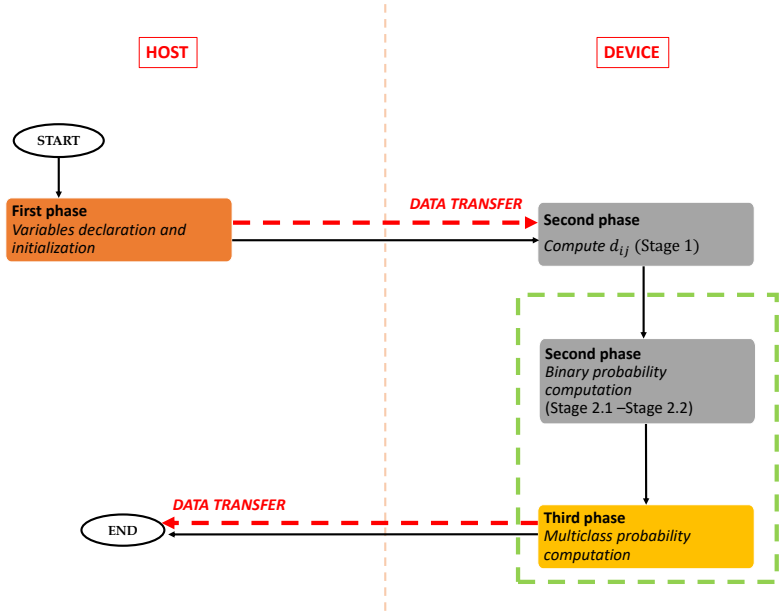


Figure 3.15: Parallel version of the SVM algorithm.

declaration and initialization are performed. Furthermore, in this phase all the data are organized to be transferred from host to device. In particular, all the outputs of the SVM training phase (which is offline and not present in this work) are stored in a linear matrix in order to perform a fast and efficient transfer. When the GPU memory is initialized and the data are transferred, the computation can start. It is important to notice that in this parallel version the *Second phase* of the algorithm is split in two steps. In particular, the computation of the distance between the sample and the hyperplane (*Stage 1*) is separated from the binary probability computation (*Stage 2.1* and *Stage 2.2*). In fact, the distance computation in the *Stage 1* is performed by applying the Equation 3.7 which can be solved by the *cublasSgemm* function of the cuBLAS library. The inputs of this function are a matrix, made up of the w vectors of all the pixels, and the input image. At this point, since both the remaining part of the *Second phase*

and the *Third phase* can be pixel-wise parallelized, it is possible to create a custom kernel that simultaneously computes the binary and the multiclass probabilities for each pixel. Even in this case the kernel is characterized by a grid whose dimension is:

$$\dim_{grid} = \frac{N}{n_{threads}} \tag{3.8}$$

where N is the number of pixels of the image and $n_{threads}$ is the number of threads that are in each block of the grid. As said before, this number is set to 32 and if the remainder of Equation 3.8 is not equal to zero, dim_{grid} is unitary incremented. This kernel starts with the binary probabilities computation (*Stage 2.1* and *Stage 2.2*) and then continues with the multiclass probabilities (*Third phase*) as described in the *Algorithm Second Phase* and in *Algorithm Third Phase* in Figure 3.14. These updated probabilities are stored in a probability map which is transferred from the device to the host.

SVM results

The values reported in Table 3.6, in the column *Time*, confirm that the serial version is well performing and, sometimes, is even faster than the parallel code. Notice that the time effectively dedicated to the SVM computation is significantly decreased in the parallel version compared to the serial one. In the column *Effective Computational Time*, the times relative only to the computation of the algorithm are presented. These values demonstrate that the parallelization of the *Second phase* and *Third phase* allow to decrease the computational time. In fact, consider that only the creation of the *handle*, used in the cuBLAS function, takes over the 28% of the entire time. Nevertheless, in the complete system development, the *handle* will be created only once for all the algorithms.

Analyzing only the time of the SVM computation, the parallel version takes only 0.009 s compared to the serial version which, instead, takes 0.233 s, if the biggest image in the database is considered (P1C2). In this case, the speed-up between the serial and the parallel SVM *effective* times is equal to 25.8x. For this reason, the parallel SVM will be also evaluated

Image ID	Version	Time	Effective Computational Time	Speed-up
P1C1	S	0.275	0.228	
	P	0.302	0.009	0.91x
P1C2	S	0.277	0.233	
	P	0.297	0.009	0.93x
P2C1	S	0.226	0.193	
	P	0.315	0.008	0.72x
P3C1	S	0.193	0.164	
	P	0.292	0.007	0.66x
P4C1	S	0.132	0.108	
	P	0.308	0.005	0.43x

Table 3.6: SVM results. These results are obtained with the *System 1*. The parallel code has been run using only one GPU of the system.

in the complete system developing in order to save time once the cuBLAS variables declaration is performed for all the algorithms. Also in this case, the algorithm has been executed on *System 2*. In Table 3.7 all the results are presented.

The same considerations can be applied to this case since the times of the serial versions are faster than the parallel ones. Despite the serial version being the best performer, if the analysis is focused only on the SVM algorithm computation, it is clear that the parallel version is significantly faster than the serial one. Also in this case, most of the computational time is dedicated to the *handle* creation (from about the 36% to the 41% considering all the images) while the percentage of time dedicated to the data transfer is very low (from about the 3% to the 6%). Concerning the classification results, it is important to notice that there are no differences in the values, and therefore all the versions classify all the pixels with the same labels. The classification results of the SVM algorithm are shown in Figure 3.16. The tumor area is represented with the red color, the green area is related to the healthy tissue, the blue indicates the hypervascularized tissue and the background is shown in black.

In order to evaluate the SVM classification results the *sensitivity*, the *specificity* and the *accuracy* metrics are used [48]. The *sensitivity* is the pro-

Image ID	Version	Time	Effective Computational Time	Speed-up
P1C1	S	0.34	0.276	
	P	0.481	0.007	0.71x
P1C2	S	0.311	0.309	
	P	0.467	0.007	0.67x
P2C1	S	0.261	0.235	
	P	0.481	0.006	0.54x
P3C1	S	0.263	0.203	
	P	0.465	0.005	0.55x
P4C1	S	0.149	0.137	
	P	0.432	0.004	0.34x

Table 3.7: SVM results. These results are obtained with the *System 2*.

portion of the actual positives that are correctly identified by the classifiers. Equation 3.9 shows that it is computed as the ratio between the *true positive* (TP) and the sum of the *true positive* and *false negative* (FN), where *true positive* means the number of real positive correctly classified, while

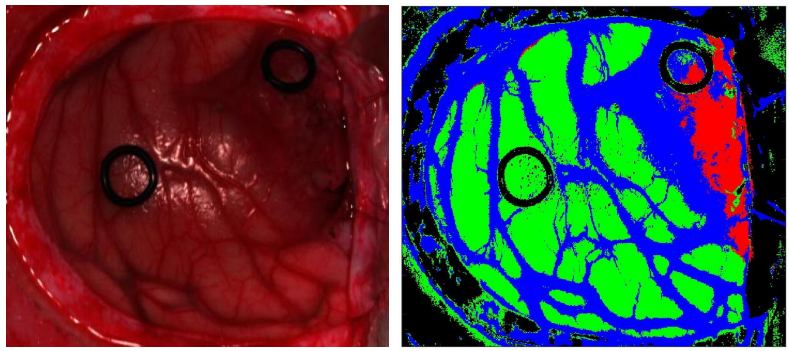


Figure 3.16: Classification results of the SVM algorithm. The image on the left (*Image ID=P2C1*) is the input of the algorithm, so the HS cube acquired during the neurosurgical operation. The image on the right is the classification map computed classifying the input with the SVM algorithm. *(Red: tumor area; green: healthy tissue; blue: hypervascularized tissue; black: background).*

the *false negative* is the number of negative samples classified as positive.

$$sensitivity = \frac{TP}{TP + FN} \tag{3.9}$$

The *specificity* is the proportion of actual negative that the classifier correctly recognizes as negative. It is the ratio between the *true negative* (TN), i.e. the number of negative samples that the classifier correctly evaluates, and the sum of *true negative* and *false positive* (FP), that is the number of samples wrongly classified as positive (Equation 3.10).

$$specificity = \frac{TN}{TN + FP} \tag{3.10}$$

Finally the *accuracy* is the ability of the classifier to correctly assign the class of a sample of unseen data. It is defined as follows:

$$accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{3.11}$$

First of all these metrics are computed providing as input the labelled samples to the classifier. Once the classifier is trained it is used to classify new data, in this case the whole HS cube. At the end, the results are evaluated by neurosurgeons in order to establish if the SVM algorithm well distinguishes the different types of tissue [48]. The values of sensitivity, specificity and accuracy of the SVM classifier are reported in [48] where also the original HELICoiD system is presented. Since the SVM classifier, described in this work, provides the same classification results of the reference system, the same values of sensitivity, specificity and accuracy are reached. In particular, considering all the images, the *accuracy* values are over 99%, and the *sensitivity* and specificity are higher than 97% and they reach 100% in the most of cases [48].

3.4.4 K-Nearest Neighbors

[1]

[1]The contents of this paragraph are published in *Florimbi G., Fabelo H., Torti E., Lazcano R., Madroñal D., Ortega S., Salvador R., Leporati F., Danese G.,*

The algorithm and the serial version

Recent uses of the KNN algorithm show that it is not restricted to a classifier role but it can also be used as a filtering technique [63] [70]. In this work, this algorithm improves the results of the spectral classification by adding spatial information. In this scope, the KNN filter receives as input the probability map Pc generated by the SVM classifier and the one-band representation I of the HS image, generated using the PCA. Considering each pixel of the HS image, the KNN performs a nearest neighbors searching step and a filtering step. Concerning the first part, the nearest neighbors of a pixel are searched in a feature space which contains the pixel values and the spatial coordinates (Equation 3.12).

$$F(q) = (I(q), \lambda \cdot l(q), \lambda \cdot h(q)) \tag{3.12}$$

where *I(q)* is the normalized pixel value of the one-band representation and *l(q)* and *h(q)* refer to the normalized coordinates of pixel q. The parameter λ controls the balance between the pixel value and the spatial coordinates. For example, the spatial information is not considered when λ is equal to zero. Otherwise, if this value is higher than zero, more influence is given to the local neighborhood [70]. The process of finding the K nearest neighbors for each pixel involves the computation of its distance from the other pixels present in the image. The Euclidean distance (Equation 3.13) is one of the metrics used in this work to evaluate the distance of a generic pixel located at (r, c) coordinates, to another located at (i, j):

$$d(I(rc), I(ij)) = \sqrt{(I_{rc} - I_{ij})^2 + (r - i)^2 + (c - j)^2} \tag{3.13}$$

where I_{rc} and I_{ij} are the normalized pixel values of the one-band representation array I at row r and column c, and at row i and column j respectively. In this work the Manhattan metric is also considered to compute distances.

Concerning the parameters presented in Equation 3.12, the distance is defined as:

$$d(I(rc), I(ij)) = |I_{rc} - I_{ij}| + |r - i| + |c - j| \tag{3.14}$$

For each pixel, the K nearest neighbors are selected after that the distances have been sorted. Once the neighbors selection is over, the filtering step can start. Taking into account the probability map Pc by the SVM classifier, a set of new optimized probabilities for each pixel is computed as shown in Equation 3.15.

$$O(q) = \frac{\sum Pc(s)}{K}, s \in w_q \tag{3.15}$$

For each pixel, the algorithm computes a number of probabilities O equal to the number of SVM classes n_{class}, that is four in this work. In Equation 3.15, w_q indicates the nearest neighbors of the pixel q, s is the index related to each neighbor and K is the number of neighbors searched for each pixel [63]. The last step consists of assigning a label to each q corresponding to the highest value among the four optimized probabilities O computed for each pixel. In this way a new classification map is computed. It is important to highlight that the K nearest neighbors selection is the most consuming part of the code from the computational point of view. For this reason, an optimization is introduced in order to save memory and execution time [63]. Instead of searching the neighbors within the entire image, the selection is done inside a search window, a region that surrounds the pixel. In this way the number of distance computations considerably decreases: in fact, if the entire image is considered, the algorithm computes *N-1* distances for each pixel, where N is the number of pixels. The choice of introducing a smaller area where to find the neighbors can be done because of the value of the parameter λ that is set for this work. Authors of [48] performed an analysis of the values of the parameters λ and K which represent the importance given to the neighborhood of the pixel and the number of nearest neighbors that have to be selected for each pixel, respectively. They concluded that small values of λ and K do not allow to represent the real distribution

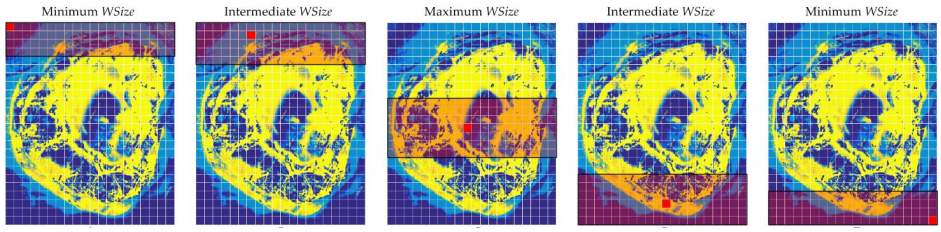


Figure 3.17: Different window sizes. The window dimension depends on the pixel location inside the image.

of the tissues. On the other hand, high values of these parameters tend to over-smooth the classes. The final values chosen are K=*40* and λ=*1*. This value of λ gives high importance to the local neighborhood. For this reason, the probability to find neighbors near the pixel is higher than the probability to find smaller distances in further zones of the image. Several analysis were performed looking for a window dimension which produces the same results as searching the neighbors in the whole image. The search window selected as reference is characterized by a number of rows (*WSize*) of the image equal to 14 and by a number of pixels equal to *14 rows*$\times$*total number of columns*. The window surrounds the pixel in a symmetric way so that half part of the area is evaluated over the pixel and the other half below it (Figure 3.17) [63]. The pixels placed near the borders are treated separately compared to the pixels in the centre of the image. To maintain a spatial coherence, the pixels present in the top part have a window with smaller dimension at first, so as not to search further than *WSize*/2 down in the image. This way, the dimension increases with each further pixel being processed until a steady state is reached. This happens when the number of pixels above the one being processed reaches *WSize*/2 and it is kept until an analogous situation happens in the lower zone of the image. The main steps of the serial version of the KNN algorithm are shown in Figure 3.18. After the declaration and the initialization of the variables (*First phase*), the algorithm computes the distances between each pixel and the ones present in the respective windows (*Algorithm Second Phase*, *lines 2 -*

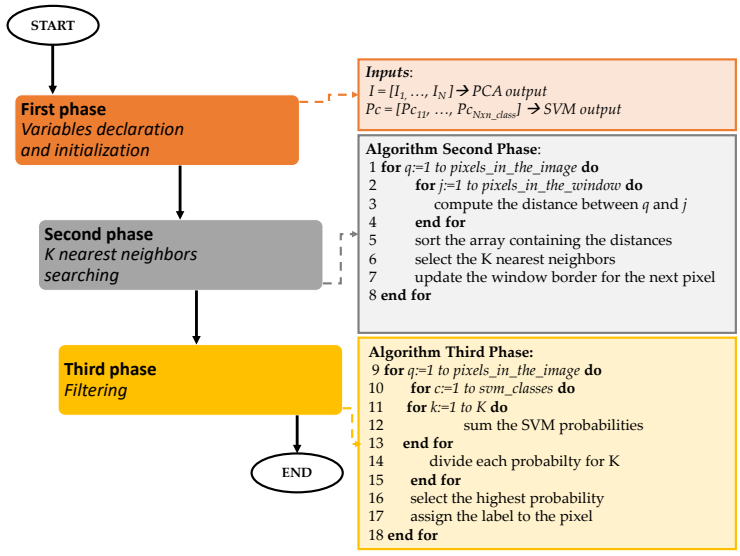


Figure 3.18: Main steps of the KNN serial flow.

4). The distances are computed using the metric shown in Equation 3.13 or in Equation 3.14 and stored in an array which is sorted in ascending order (*line 5*). In this way, the pixels related to the first K elements (distances) in the array are the K selected neighbors (*line 6*). At the end of this *Second phase*, the window parameters are updated for the computations related to the next pixel (*line 7*). Once the pixel neighbors are selected, the *Third Phase* can start. The goal of this algorithm is to assign a label to each pixel considering the probability map generated by the SVM classifier. For each pixel, the algorithm computes a number of probabilities $O(q)$ (Equation 3.15) equal to the number of classes. In particular, for each class, it sums the probabilities of the neighbors and, then, it divides the results by K (*Algorithm Third Phase*, *lines 10-15*). After computing the four optimized probabilities for each pixel, the algorithm assigns the label corresponding

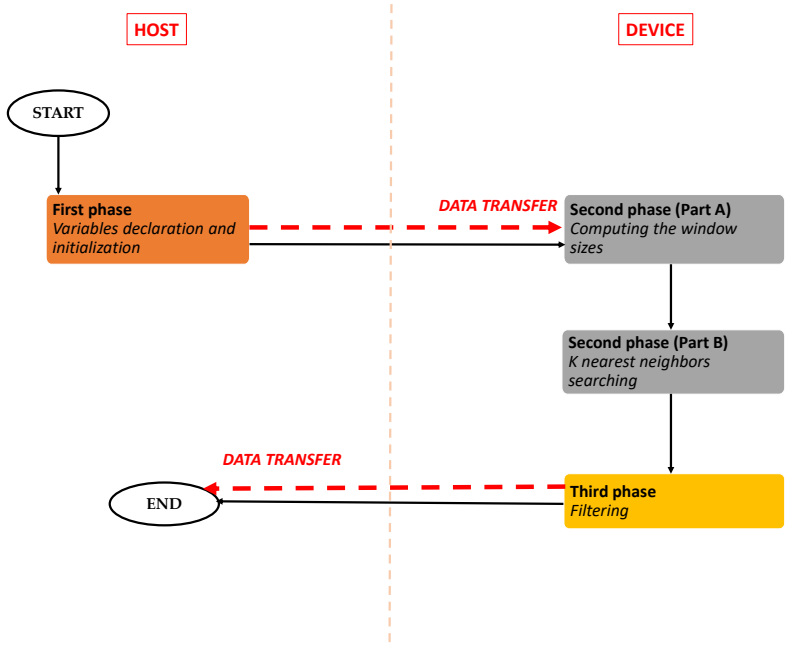


Figure 3.19: KNN parallel version.

to the highest probability that the pixel has to belong to the class (*lines 16-17*).

The parallel version

The basic idea followed in the development of the parallel version is that each CUDA core has to assign a label to each pixel simultaneously. The flow starts on the host with the declaration and initialization of all the variables (*First phase* in Figure 3.19). The main difference between this *First phase* and the corresponding one of the serial code is that, in this parallel implementation, the number of arrays, structures and variables allocations is decreased in order to save memory. After the *First phase*, the algorithm transfers to the device the one-band representation generated by the PCA algorithm and the probability maps generated by the SVM

classifier. The flow proceeds with the resources allocation on the device. The first step of the KNN filtering algorithm on the GPU device concerns the execution of a kernel that evaluates the borders and the size of the windows in parallel through the pixels (*Second phase (Part A)*). Contrary to the serial code execution, where the parameters related to the window dimensions are updated at the end of the neighbors selection for each pixel, in the parallel version the algorithm needs to know these variables before starting the KNN filtering computation. In fact, in the following steps, it is important to copy the PCA and SVM data (already transferred to the device) from the global to the local memory of the GPU, shared by the threads within a block. For this reason, each thread copies the part of the data (delimited by the window parameters) needed in the computation. Then, the results are copied to the global memory only at the end of the kernel execution. This step is crucial to decrease the execution time since the accesses to the global memory are very slow. In the *Second phase (Part B)* each thread evaluates the K nearest neighbors of a pixel in parallel. First, the PCA data required by each block are copied from global to the local memory. Then, each thread of the block declares an array called *neighbors distances*, whose dimension is equal to the number of K neighbors, which is set to 40 in this work. This array is initialized with large values and will contain the 40 lowest distances computed between the pixels. The implementation goes on computing the distance between pixel i, represented by the thread, and all the pixels within its window. If the distance between the pixel i and a pixel j inside the window is smaller or equal to the last element of the *neighbors distances* array, this distance will be stored in the last position of the array. It is important to highlight that this *if* statement is always verified considering the first 40 pixels in the window (i.e., the first 40 for loop iterations). Once the first 40 iterations are executed, in the last position of the array there will be a real distance (not the initialization value) and it will be the highest value among those already present in the array. This is verified because every time that a new distance is stored in the array, the algorithm calls a sort function in order to sort the elements of the array in an ascending order, keeping track of their indexes. The K indexes of the selected neighbors are the output of the kernel and will be copied to

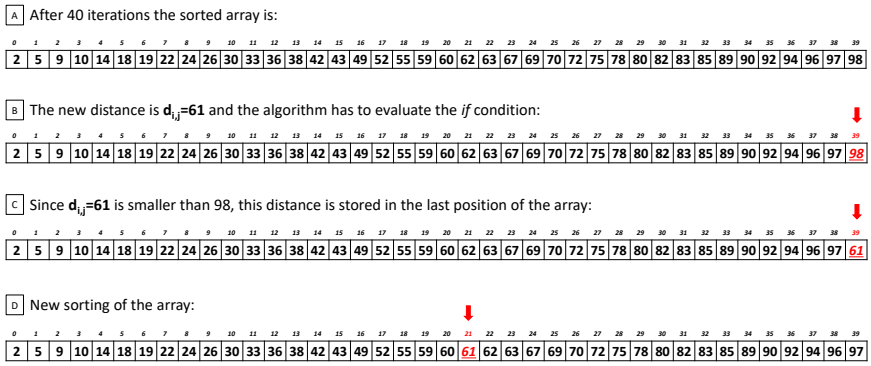


Figure 3.20: KNN searching new distance evaluation example.

the global memory. Figure 3.20 shows an example of the evaluation of a new distance by the KNN searching algorithm. After the first 40 iterations, the array contains 40 distances stored in ascending order (Figure 3.20.A). When a new distance is computed (in this example its value is 61), it is compared with the last element of the array, in this case located in position 39 and whose value is 98 (Figure 3.20.B). Since the new distance is lower than 98, it is stored in the last position of the array (Figure 3.20.C). At this point, the array is sorted again (Figure 3.20.D). Due to the reduced dimension of the array to be sorted, the sort function implemented in this work is the shell sort algorithm. After computing all the neighbors for all the pixels, the *Third phase* step of the KNN algorithm starts. In this phase, the KNN filtering is computed by every thread of each block. First, each thread copies the SVM probabilities of their corresponding neighbors from the global to the local memory. The pseudo-code of the *Third Phase* is presented in Algorithm 6.

```
1  transfer the SVM data from the global to the local memory
2  declare and initialize the variables max_probability and label
3  for class ← 1 to svm_class do
4  |  initialize temporary_probability to zero
5  |  for k ← 1 to K do
6  |  |  compute temporary_probability
7  |  |  if class = 1 then
8  |  |  |  max_probability=temporary_probability
9  |  |  |  label=class
10 |  |  else
11 |  |  |  if temporary_probability > max_probability then
12 |  |  |  |  max_probability=temporary_probability
13 |  |  |  |  label=class
14 |  |  |  end
15 |  |  end
16 |  end
17 end
```

Algorithm 6: *Third Phase* pseudocode.

For each class, the algorithm computes the *temporary_probability* value of each pixel, which is the sum of the SVM probabilities of all the neighbors of the reference pixel (*lines 3-6*). If the algorithm is executing the first iteration of the first for loop (i.e., if it is considering the first class), the variable *max_probability* assumes the value of *temporary_probability* variable and the index of the class is stored in label (*lines 7-10*). In the following iterations, after computing the *temporary_probability*, its value is stored only if it is higher than the *max_probability* value (which represents the highest probability value of the previous classes). In this case, the index of the class is also stored (*lines 11-14*). At the end of the *for* loop that iterates on the number of classes, the algorithm selects the pixel label corresponding to the highest probabilities sum among the four classes. It is worth noting that the algorithm immediately evaluates if the sum of probabilities could be the highest among the classes or not. This fact means that some arrays declared in the serial version can be replaced with a few variables, thus saving memory. At the end of this phase, the label of the pixel, which is the output of this step, is stored in the global memory. Also in this case, for each kernel, the block dimension is equal to 32. As said before, this choice

is related to the warp dimension defined by the CUDA framework. Once the KNN algorithm execution ends on the GPU device, an array containing the labels of all the pixels is transferred from the GPU device to the CPU host. At this point, the memory can be released.

KNN results

This paragraph presents the results of the KNN-based filtering algorithm analyzing the computational times of the serial and the parallel versions. Furthermore, these times will be analized considering the evaluation of different window sizes and of different metrics to compute the distances. As said in the previous paragraphs, an important optimization is introduced in the serial and parallel implementations concerning the computation of the distances between pixels inside a window and not within the entire image. Reducing the space where the algorithm evaluates the distances ensures a significant decrease of the computational time, as shown in Table 3.8. In particular, the table provides the execution times for all the images, considering both the cases in which the neighbors are searched within the entire image (*EI*) and within a window of 14 rows (*WSize14*). The speed-up obtained with the optimization has been also included. In addition, this table shows the total number of pixels of each image and the number of pixels inside the smallest and the biggest window in the *WSize14* implementation. The times refer to tests where the Euclidean distance has been considered. The simulations of the serial code have been carried out on the Intel i7 processor which belongs to the *System 1*.

Data presented in Table 3.8 show that this optimization allows a huge decrease in the execution times. For example, considering the biggest image of the dataset, P1C2, the time of the implementation for the entire image is 19,135.58 s (about 5 h and 30 min). On the contrary, considering a window of 14 rows, this time decreases to 509.16 s (about 8 min). The reason of this huge time difference is that, when the algorithm has to consider the entire image, it needs to compute a number of distances equal to (264,408 – 1) for each pixel, where 264,408 is the number of pixels of the P1C2 image. However, with the window technique, the algorithm computes a number of

Image ID	#Pixels	*EI* Time [s]	*WSize14* Time [s]	Speed-up	Min *WSize14* [#Pixels]	Max *WSize14* [#Pixels]
P1C1	251,532	17,173.74	503.89	34.08x	3,836	7,672
P1C2	264,408	19,135.58	509.16	37.58x	3,864	7,728
P2C1	219,232	15,630.77	374.67	41.72x	3,472	6,944
P3C1	185,368	9788.58	322.86	30.32x	3,451	6,902
P4C1	124,691	4015.89	139.30	28.83x	2,303	4,606

Table 3.8: KNN results considering both the entire image and the reference window. Execution times of the serial code considering as search space both the entire image (*EI* Time) and a window with 14 rows (*WSize14* Time).

distances that, for the same image, varies from (3,864 – 1) to (7,728 – 1), where 3,864 and 7,728 are the numbers of pixels inside the windows with the minimum and the maximum sizes, respectively (depending whether the pixel is in the borders or in the center of the image). It is important to highlight that the highest speed-up is achieved analyzing the image P2C1, since it has a number of pixel which is an integer multiple of 2, allowing better use of the CPU resources and faster memory accesses. Concerning the classification results, notice that there are no differences in the results, and therefore all the pixels are classified with the same labels, using either the entire image or a window. Considering this significant result, the computational time variations are evaluated when the window size is reduced. Furthermore, since the main goal of the work is to reach the real-time execution, a parallel version of the algorithm has been developed in CUDA language to exploit the GPU technology. Table 3.9 shows the execution times of the serial and parallel implementations characterized by window sizes that vary from 14 to 2 with decrements of 2. In addition, the speed-up between the serial and the parallel codes (executed onto the Tesla K40 GPU) are presented.

The reduction of the window size supposes a decrease in the execution times because the algorithm has to compute a lower number of distances. For example, considering the P1C2 image, the time varies from 509.16 s (∼8 min) to 62.36 s (∼1 min) in the serial versions of *WSize14* and *WSize2*, respectively. If the parallel implementation of the same image is considered, the times present a further decrease. In fact, for the same image the paral-

Image ID	Version	*WSize14* Time [s]	*WSize12* Time [s]	*WSize10* Time [s]	*WSize8* Time [s]	*WSize6* Time [s]	*WSize4* Time [s]	*WSize2* Time [s]
P1C1	Serial	503.89	406.32	383.71	262.00	221.71	118.25	59.08
	CUDA	12.83	11.49	6.23	5.52	3.85	2.29	1.22
	speed-up	**39.25x**	**35.33x**	**61.59x**	**47.42x**	**57.52x**	**51.44x**	**48.10x**
P1C2	Serial	509.16	424.22	408.52	276.06	235.76	125.34	62.36
	CUDA	13.53	12.09	6.47	5.73	3.99	2.39	1.26
	speed-up	**37.62x**	**35.08x**	**63.07x**	**48.11x**	**59.04x**	**52.31x**	**49.15x**
P2C1	Serial	374.67	315.73	302.54	239.58	151.23	95.02	47.06
	CUDA	10.55	5.70	5.18	3.62	2.67	1.70	1.03
	speed-up	**35.51x**	**55.39x**	**58.30x**	**66.18x**	**56.58x**	**55.76x**	**45.62x**
P3C1	Serial	322.86	263.40	254.30	202.56	122.16	78.47	39.97
	CUDA	9.00	4.92	4.45	3.15	2.30	1.51	0.92
	speed-up	**35.85x**	**53.46x**	**57.06x**	**64.17x**	**52.92x**	**51.80x**	**43.16x**
P4C1	Serial	139.30	118.94	115.07	90.81	55.29	35.84	18.11
	CUDA	3.21	2.34	2.16	1.63	1.12	0.83	0.60
	speed-up	**43.38x**	**50.63x**	**53.23x**	**55.61x**	**49.01x**	**42.82x**	**30.13x**

Table 3.9: KNN results considering different window sizes. Execution time results of the serial and parallel implementations using the Euclidean distance employing different window sizes.

lel version of *WSize14* is ∼37× times faster than the serial version, taking only 13.53 s instead of ∼8 min. At the same time, the parallel execution of *WSize2* takes only 1.26 s instead of ∼1 min (the speed-up is ∼49×). Concerning all the images in the reference implementation *WSize14*, the speed-up are always higher than 35× and, in the best case (P4C1), it reaches 43×. If we consider all the other versions, with the decreased windows sizes, the parallel code shows even higher speed-up. For example, considering the P1C2 image and the window size *WSize10*, the parallel code takes 6.48 s while the serial version takes 408.52 s (∼6 min), obtaining a speed-up of ∼63×. Nevertheless, it is necessary to examine these times and speed-up also taking into account the classification results. It is very important to consider if, when reducing the window size, there are pixels classified with different labels compared to the *reference* version (*WSize14*). Table 3.10 shows the number of misclassified pixels between the reference result and other window sizes. Additionally, the difference percentage is shown. Considering the first three windows sizes (*WSize12*, *WSize10*, *WSize8*) for all the images, the number of pixels classified with different labels is very low, taking into account the final application of the system. In fact, the highest

Image ID	#Pixels	*WSize12*	*WSize10*	*WSize8*	*WSize6*	*WSize4*	*WSize2*
P1C1	251,532	0 (0.000%)	0 (0.000%)	65 (0.026%)	3,672 (1.460%)	9,096 (3.616%)	20,476 (8.141%)
P1C2	264,408	0 (0.000%)	1 (0.000%)	60 (0.023%)	2,845 (1.076%)	7,705 (2.914%)	22,054 (8.341%)
P2C1	219,232	0 (0.000%)	4 (0.002%)	65 (0.030%)	2,606 (1.189%)	6,532 (2.979%)	18,015 (8.217%)
P3C1	185,368	0 (0.000%)	1 (0.000%)	49 (0.026%)	2,273 (1.226%)	5,604 (3.023%)	13,981 (7.542%)
P4C1	124,691	3 (0.002%)	7 (0.005%)	71 (0.057%)	1,498 (1.201%)	3,733 (2.933%)	10,089 (8.091%)

Table 3.10: Classification differences. Number of pixels with different classification results using the Euclidean distance between the different computed windows sizes and the reference one (*WSize14*). The table shows also the difference percentages.

percentage of different pixels is 0.057%, and it is related to the P4C1 image, which, in the *WSize8* version, presents 71 different pixels on a total amount of 124,691 pixels. Concerning the other three windows sizes, the highest percentage of different pixels for window *WSize6* is 1.46% considering the P1C1 image (3,672 different pixels on 251,532). For the window *WSize4*, the percentage of different pixels is 3.62%, referred also to the P1C1 image (9,096 different pixels on 251,532) and for *WSize2*, the highest percentage is ∼8.341%, considering the biggest image of the database, P1C2 (22,054 different pixels on 264,408). At this point, there is a further evaluation that can be made considering that this algorithm is part of a system whose main goal is to discriminate between tumor and healthy tissue. The classification is made between four classes that are normal tissue, tumor tissue, hypervascularized tissue and background [48]. From the surgical and medical point of view, it is clear that a wrong discrimination between tumor and healthy tissue has much greater and transcendental relevance than just a misclassification issue between tumor and any other classes (hypervascularized and background) or between healthy, hypervascularized and background classes. It is possible to re-evaluate again the results of Table 3.10, considering that, in the different *WSize* executions, only a low percentage of different pixel labels are exchanged between tumor and normal tissue. Figure 3.21 shows the percentage of pixels that are misclassified between tumor and healthy tissues, tumor and hypervascularized tissues and tumor and background, respectively, considering all the windows sizes for each image compared to the reference version. In addition, the graph presents the classification differences between healthy, hypervascularized and background classes (called *Others*). As it can be seen in Figure 3.21, only in the case of the P3C1 im-

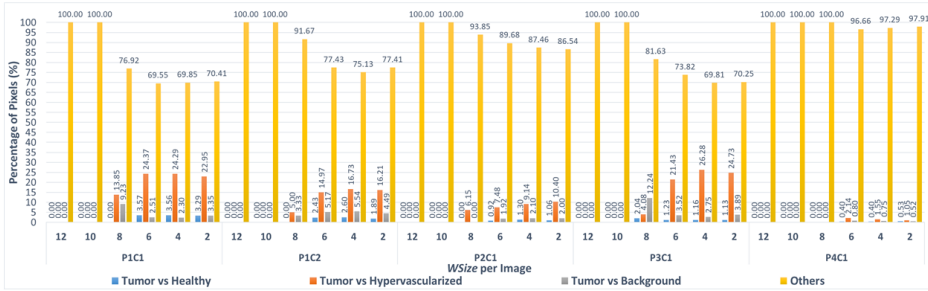


Figure 3.21: KNN classification results. Percentage of pixels that have been misclassified using the Euclidean distance between tumor and healthy tissues (blue), tumor and hypervascularized tissues (orange), tumor tissue and background (gray) and the other misclassifications between healthy, hypervascularized and background (yellow). The results were obtained per each window size implementations compared to the *WSize14* for each image of the dataset.

age using the *WSize8*, the algorithm misclassifies approximately 2% of the pixels (1 out of 49 pixels), exchanging the labels between tumor and healthy tissues. In all the other implementations of *WSize8*, the classification differences do not involve the tumor class. Furthermore, in the versions related to the three smallest windows (*WSize6*, *WSize4*, *WSize2*), the percentage of the pixels exchanged between these two classes is lower than the percentages of pixels exchanged between the other classes. For example, for the biggest image of the database (P1C2), in the *WSize6* implementation, the classification difference between tumor and healthy tissue represents 2.43% out of 2,845 different pixels. Considering the same image in the *WSize4* and in the *WSize2* implementations, this percentage is 2.60% out of 7,705 pixels and 1.89% out of 22,054 pixels, respectively. The highest percentage of differences between these two classes is found in the *WSize6* version regarding the P1C1 image, where it is around 3.57% out of 3,672 pixels. According to these data, it is clear that the algorithm can correctly distinguish the tumor from the healthy tissue, while it makes more errors in separating the tumor from the hypervascularized tissue. The highest percentages of misclassified pixels between the tumor and the hypervascularized classes reach 26.28% of the total number of different pixels (P3C1 image, *WSize4* version). In

	P1C1		P1C2		P2C1		P3C1		P4C1	
Distance Type	**EI**	**WSize14**	**EI**	**WSize14**	**EI**	**WSize14**	**EI**	**WSize14**	**EI**	**WSize14**
Euclidean [s]	17,173.75	503.89	19,135.58	509.17	15,630.77	374.67	9,788.58	322.87	4,015.89	139.30
Manhattan [s]	7,222.13	190.02	7,683.44	202.42	4,735.87	146.93	3,382.84	121.37	1,807.91	55.91
Speed-up	2.38x	2.65x	2.49x	2.51x	3.3x	2.55x	2.89x	2.66x	2.22x	2.49x
	1.33%		0.99%		1.03%		1.10%		1.06%	

Table 3.11: Comparison between the Euclidean and the Manhattan metrics. Comparison of the execution times of the serial versions obtained employing the Euclidean and Manhattan distances with the entire image (*EI*) and the *WSize14*. The table also presents the classification differences between the Euclidean and Manhattan implementations.

fact, according to what it is said in [48], these two classes referred to tissues with similar spectral signatures that can produce some misclassifications. On the other hand, the spectral signatures of tumor and healthy tissues present remarkable differences that allow the algorithm to distinguish these two classes in the classification.

Another technique adopted to further reduce the execution time of the distances computation is the use of the Manhattan metric (Equation 3.14) instead of the Euclidean one. Table 3.11 compares the times of the serial code using both the entire image (*EI*) and the reference window (*WSize14*), applying both the Euclidean and the Manhattan distances. The speed-up obtained using the Manhattan distance and the percentages of pixels that are different in the results are also presented in this table. A further reduction of the execution time is obtained using the Manhattan metric in the distance computations (Table 3.11). In fact, for the biggest image of the database (P1C2), the time is reduced from ∼5 h (19,135.58 s) using the Euclidean distance to ∼2 h (7,683.44 s) in the case of using the entire image. If the neighbors are searched within the window (for the same image), the time decreases to ∼3 min (202.42 s) using the Manhattan distance. Concerning all the images, it is possible to reach speed-up from 2.22× to 3.33×, considering the versions with the entire image, and from 2.49× to 2.66× in the *WSize14* executions. Comparing the implementations that exploit the Manhattan distance and the ones that use the Euclidean metric, the number of pixels classified with different labels is quite low: the highest percentage of different pixels is 1.33% in the P1C1 image. Furthermore, consider that there are no differences in the classification results comparing the entire image and the *WSize14* versions, using the Manhattan distance.

Image ID	Version	*WSize14* Time [s]	*WSize12* Time [s]	*WSize10* Time [s]	*WSize8* Time [s]	*WSize6* Time [s]	*WSize4* Time [s]	*WSize2* Time [s]
P1C1	Serial	190.02	192.19	158.15	129.54	81.61	54.14	28.70
	CUDA	7.63	7.07	5.04	4.62	3.38	2.04	1.18
	Speed-up	**24.90x**	**27.15x**	**31.34x**	**27.99x**	**24.09x**	**26.47x**	**24.13x**
P1C2	Serial	202.41	204.65	169.23	138.12	84.98	57.51	29.75
	CUDA	8.01	7.40	5.21	4.84	3.51	2.09	1.22
	Speed-up	**25.26x**	**27.64x**	**32.48x**	**28.52x**	**24.20x**	**27.45x**	**24.28x**
P2C1	Serial	146.92	152.20	125.60	102.94	63.90	42.57	21.81
	CUDA	6.44	4.58	4.27	3.16	2.34	1.52	1.00
	Speed-up	**22.81x**	**33.20x**	**29.35x**	**32.50x**	**27.25x**	**27.86x**	**21.66x**
P3C1	Serial	121.37	126.98	104.79	86.83	55.09	36.24	18.54
	CUDA	5.57	4.03	3.76	2.79	2.04	1.37	0.90
	Speed-up	**21.75x**	**31.47x**	**27.86x**	**31.11x**	**26.88x**	**26.42x**	**20.54x**
P4C1	Serial	55.91	58.06	42.62	39.42	24.39	16.62	8.66
	CUDA	2.81	2.12	1.98	1.49	1.04	0.80	0.60
	Speed-up	**19.87x**	**27.27x**	**21.46x**	**26.46x**	**23.31x**	**20.59x**	**11.42x**

Table 3.12: KNN results considering different window sizes. Execution time results of the serial and parallel implementations using the Manhattan distance and using different window sizes.

At this point, it is interesting to evaluate how the execution times can be reduced changing the size of the windows using the Manhattan metric in the distances computation. The results shown in Table 3.12 confirm that decreasing the number of distance computations, i.e. the variations of the window sizes, allows further reductions of the computational time. The lowest execution times are obtained exploiting the GPU technology that can run the parallel algorithm taking ∼8 s (compared to ∼3 min) if the biggest image (P1C2) with the *WSize14* version is considered. The speed-up obtained using this device and the optimizations introduced in the code are significant and they can reach up to 33.2× (P2C1– *WSize12*). For some images and for some window dimensions, the algorithm takes only a few seconds, but what is even more important to consider is the number of pixels that are misclassified when the window size decreases (Table 3.13). As it can be seen in the results shown in Table 3.13, *WSizes12* and *WSizes10* present a reduced number of different pixels compared to the other implementations. Analyzing the Euclidean distance results presented in Table 3.10, this consideration can be made for the first three tests (*WSizes12*, *WSizes10* and *WSizes8*) but, in this case, the number of different pixels in *WSizes8* is higher than the first two versions. Despite this, it is important

Image ID	#Pixels	*WSize12*	*WSize10*	*WSize8*	*WSize6*	*WSize4*	*WSize2*
P1C1	251,532	2 (0.001%)	65 (0.026%)	1,832 (0.728%)	4,507 (1.792%)	8,889 (3.534%)	19,783 (7.865%)
P1C2	264,408	3 (0.001%)	56 (0.021%)	1,483 (0.561%)	3,891 (1.471%)	7,839 (2.965%)	21,714 (8.812%)
P2C1	219,232	3 (0.001%)	44 (0.020%)	1,320 (0.602%)	3,483 (1.589%)	6,743 (3.075%)	17,653 (8.052%)
P3C1	185,368	2 (0.001%)	48 (0.026%)	1,033 (0.557%)	2,825 (1.524%)	5,474 (2.953%)	13,436 (7.248%)
P4C1	124,691	2 (0.001%)	35 (0.028%)	599 (0.560%)	2,014 (1.615%)	3,831 (3.072%)	9,613 (7.709%)

Table 3.13: Classification differences. Number of pixels with different classification results using the Manhattan distance between the computed window sizes and the *WSize14* Manhattan.

to highlight that the classification differences shown in Table 3.13 are not very relevant for the final application of the system. In this application, a solution with a good compromise between real-time execution and classification accuracy of the results has to be selected. In addition, it is also important to evaluate the percentage of different pixels that are misclassified between tumor and healthy tissues and between tumor and the other classes. Figure 3.22 shows the percentage of pixels that are misclassified using the Manhattan metric between the different classes. In this figure, it is possible to notice that the algorithm misclassifies more pixels between tumor and hypervascularized classes than between tumor and healthy classes. In fact, the highest percentage of pixels misclassified is 30.77% related to the P1C1 image with *WSize10*, where the algorithm exchanges the labels of 20 pixels (between tumor and hypervascularized tissue) out of a total amount of 65 different pixels compared to the reference version *WSize14* (Table 3.13). Concerning the comparison between tumor and healthy classes, the number of pixels with an exchanged label is very low: the worst case is always the P1C1 image (*WSize8*), where 66 out of 1832 different pixels are misclassified, being a 3.60% of pixels.

Once presented all the results, it is important to select the suitable version for this application that is the option that offers a good compromise between an accurate classification and a fast execution. For this reason, these results should be graphically analized in order to visually appreciate the classification of the image. In Figure 3.23, the KNN filtered maps obtained from the P2C1 image and the binary maps are shown, where the differences between the evaluated window size version and the reference version are highlighted.

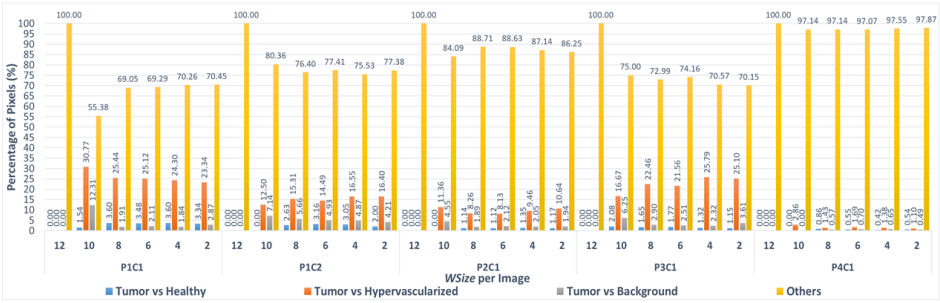


Figure 3.22: KNN classification results. Percentage of pixels that have been misclassified using the Euclidean distance between tumor and healthy tissues (blue), tumor and hypervascularized tissues (orange), tumor tissue and background (gray) and the other misclassifications between healthy, hypervascularized and background (yellow). The results were obtained per each window size implementations compared to the *WSize14* for each image of the dataset.

As described in Table 3.10, by using the window sizes *WSize12*, *WSize10* and *WSize8*, the number of different pixels was lower than those obtained using the *WSize6*, *WSize4* and *WSize2*, compared with the reference test (*WSize14*). Despite this, it is possible to see that also the *WSize8* and

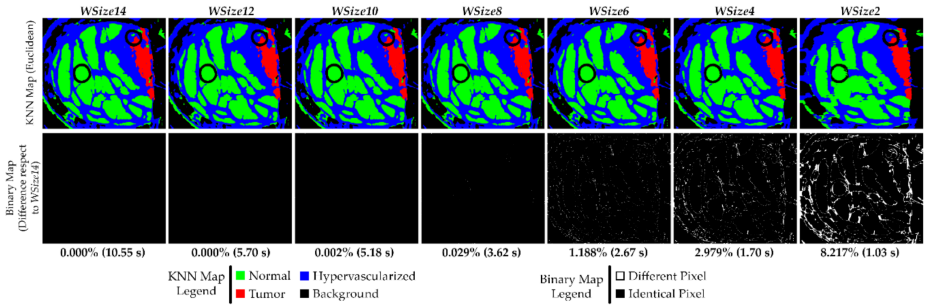


Figure 3.23: KNN classification results. Results of the KNN filtering algorithm obtained from the P2C1 image using the Euclidean distance. The first row shows the filtered classification maps generated using different window sizes. The second row presents the binary maps where the pixel differences between the current generated map and the reference one (WSize14) are shown. In addition, the percentage of differences and the execution time results are detailed.

the *WSize6* KNN filtered maps do not present relevant dissimilarities. In fact, it is important to remember that the main goal of these maps is to delineate the tumor area, in order to provide the surgeons with a guidance tool during the tumor resection. In this context, it is clear that the number of different pixels in *WSize8* and *WSize6* versions are not so significant for the final application of the system, since the surgeon always resects a security margin around the tumor tissue. As said before, this algorithm well discriminates the tumor from the healthy tissue and this consideration can also be seen in the KNN filtered maps, where the area related to the tumor tissue (red) remains roughly the same in the implementations *WSize6*, *WSize8*, *WSize10* and *WSize12* compared to the *WSize14* one. Considering the *WSize4* and the *WSize2* versions, it is possible to appreciate that the margins of the tumor are not as evident and well defined as in the other images, confirming what has been said in the previous paragraphs while analyzing the classification results. The binary maps, presented in the second row of the Figure 3.23, show the pixel differences between all the window size versions and the reference implementation (*WSize14*). In particular, by analyzing the binary maps of *WSize4* and *WSize2*, it is possible to identify several differences compared to *WSize14*. For this reason, these two versions should not be chosen for the final solution. However, in the binary maps of *WSize6* and *WSize8*, there are few differences, and they are barely appreciated analyzing the KNN filtered maps. Considering the computational times of these two parallel versions, *WSize8* takes ∼3.62 s to filter the P2C1 image, while the *WSize6* implementation is executed in ∼2.67 s. For the biggest image of the database (P1C2), the *WSize6* implementation allows to save ∼2 s when compared to the *WSize8* version. According to these results, the *WSize8* version has been selected as the best solution, giving priority to the classification accuracy but considering also its fast implementation. On the contrary, the *WSize6* implementation has been chosen as the fastest implementation with acceptable accuracy results. Similarly, the same evaluation can be done considering the implementations that exploit the Manhattan metric for the computation of the distances. The first row of the Figure 3.24 presents the KNN filtered maps of the P2C1 image. The second row presents the binary maps to evalu-

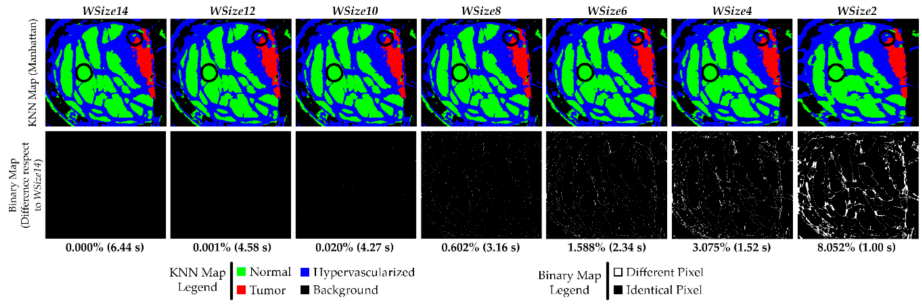


Figure 3.24: KNN classification results. Results of the KNN filtering algorithm obtained from the P2C1 image using the Manhattan distance. The first row shows the filtered classification maps generated using different window sizes. The second row presents the binary maps where the pixel differences between the current generated map and the reference one (*WSize14*) are shown. In addition, the percentage of differences and the execution time results are detailed.

ate the differences between the developed versions when compared to the *WSize14* implementation. According to the data presented in Table 3.13, it is possible to notice that the KNN filtered maps of the *WSize12* and of the *WSize10* versions are pratically identical to the *WSize14* map. Also their binary maps show that these implementations offer the highest accuracy but the slowest execution times. From the version *WSize8* to the version *WSize2* the number of misclassified pixels drastically increases. Focusing on the *WSize4* and *WSize2* maps, it is possible to notice that the border of the tumor are not well delineated. In fact, also their binary maps show a high amount of different pixels. Even if they are the fastest implementations, they can not be chosen for this application. Concerning the *WSize6* and the *WSize8* filtered and binary maps, the classification differences are not so evident, especially taking into account the tumor tissue area. As presented before, the *WSize8* version has the highest accuracy but the execution time is slower than the *WSize6* version (the former exhibits 3.16 s and the latter 2.34 s). Also in this case, the best solution is chosen on the basis of the degree of accuracy and the time constraints that the application requires. Once selected the best versions both in the Euclidean and the Manhattan

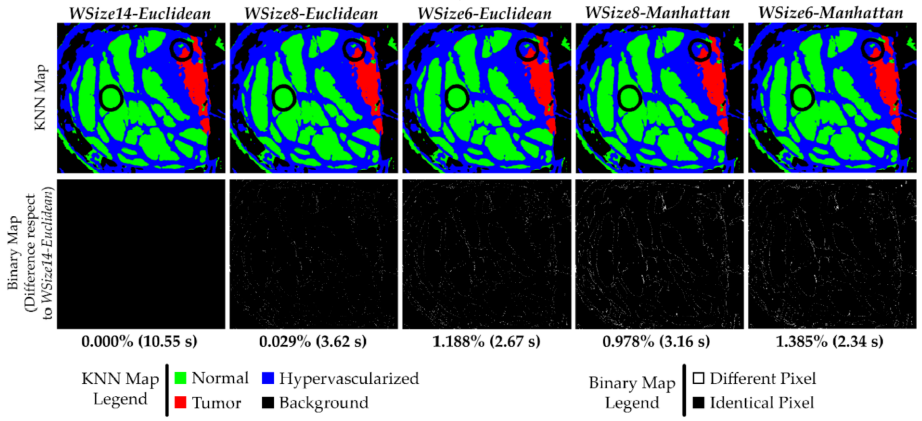


Figure 3.25: Comparison between the best solutions. Results comparison of the KNN filtered maps from the P2C1 image using both the Manhattan and Euclidean distances. The first row shows the filtered classification maps generated using different window sizes and distance metrics. The second row presents the binary maps where the pixel differences between the current generated map and the reference one (*WSize14*-Euclidean) are shown. In addition, the percentage of differences and the execution time results are detailed.

implementations, it is important to visually compare their classification results with the KNN filtered map of the *WSize14* Euclidean version. This is considered the *reference* because it does not present classification differences with the original version, which considered the entire image. For this reason, Figure 3.25 shows the comparison of the KNN filtered and binary maps between the *reference* and the *WSize8* and *WSize6* versions, cosidering both the Euclidean and the Manhattan metrics. In Figure 3.25 is possible to notice that in all the obtained KNN filtered maps, the boundaries of the tumor area are accurately defined. The solutions where the results are more similar to the reference implementation are the *WSize8 – Euclidean* and *WSize8 – Manhattan* versions, which differ 0.029% and 0.978% respectively, compared to the *WSize14 – Euclidean* reference. The versions characterized by a window with 6 rows are less accurate than the previous ones, but they are faster. Concerning the computational times, the parallel execution of the reference solution is executed in ∼10.55 s, while

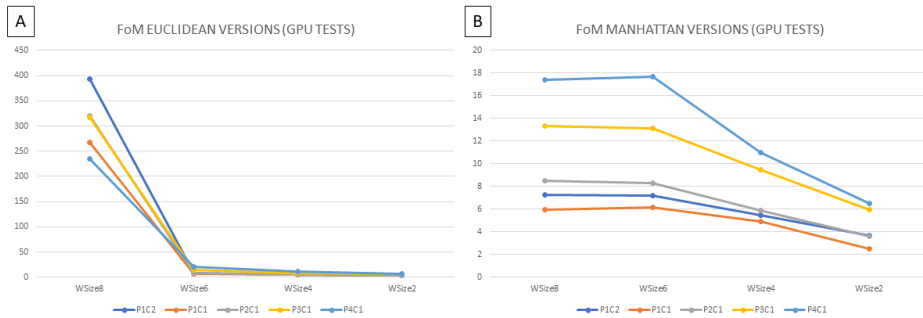


Figure 3.26: Comparison between the best solutions. Figure of metric computed comparing (A) the Euclidean versions *WSize8*, *WSize6*, *WSize4* and *WSize2* with the reference *WSize14-Euclidean*, (B) the Manhattan versions *WSize8*, *WSize6*, *WSize4* and *WSize2* with the reference *WSize14 – Euclidean*.

the *WSize8 – Euclidean* and *WSize8 – Manhattan* versions are executed in 3.62 and 3.16 s, respectively. The *WSize6 – Euclidean* and the *WSize6 – Manhattan* implementations require 2.67 and 2.34 s, respectively. Finally, a figure of merit (*FoM* in Equation 3.16), which relates the execution time (t) and the classification results (err), was considered to select the best solution that offers the highest value:

$$FoM = \frac{1}{t \times err} \tag{3.16}$$

The version *WSize8 – Euclidean* is chosen as the best solution since it presents the highest value of *FoM* (Figure 3.26). The results obtained in this analysis show that, for the proposed final application, the implementation characterized by a search window of 8 rows using Euclidean distance is the best solution. This version performs the classification of the considered images in less than 6 s, with speed-up up to 102.5× and 4317.9× compared with the *WSize14 – Euclidean* and the *entire image* versions, respectively. This version will be used in the complete system.

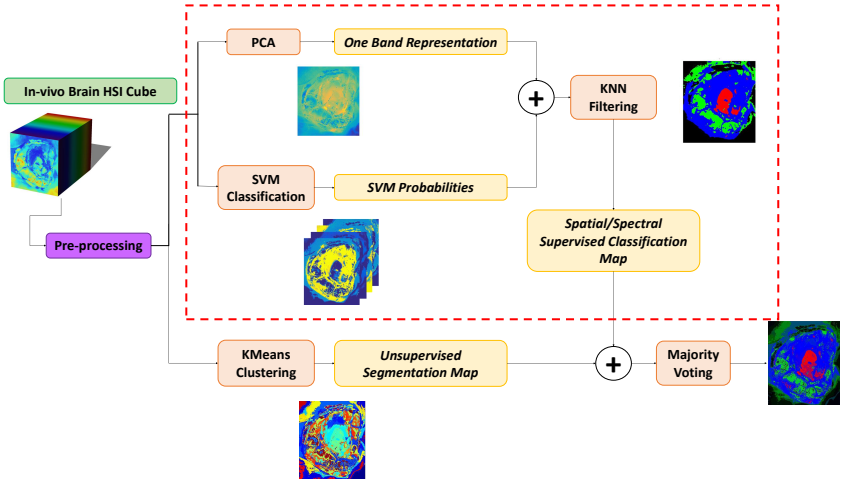


Figure 3.27: Spatial-spectral supervised classification system. The PCA, SVM and KNN algorithms represents the Spatial-spectral classification system which is the part of the complete system which performs a supervised classification.

3.4.5 Spatial-spectral supervised classification

As already said at the beginning of the paragraph 3.4, the Principal Component Analysis, the Support Vector Machine and the K Nearest Neighbors consitute the Spatial-spectral supervised classification system (in the red frame in Figure 3.27) which is the main part of the brain cancer detection system. As described before, different versions of each algorithm have been developed. They have been also included in the design of different versions of the Spatial-spectral classification system in order to find the most efficent solution. First of all a *Serial Version* of the Spatial-spectral supervised classification system has been developed. It is characterized by the sequential execution of all the algorithms in cascade on the CPU.
The first parallel code implementation is presented in Figure 3.28. As it is possible to see in the figure, the flow starts on the host where the input data are read and stored in suitable variables. These inputs are the HS image and all the variables related to the pre-processing and the different

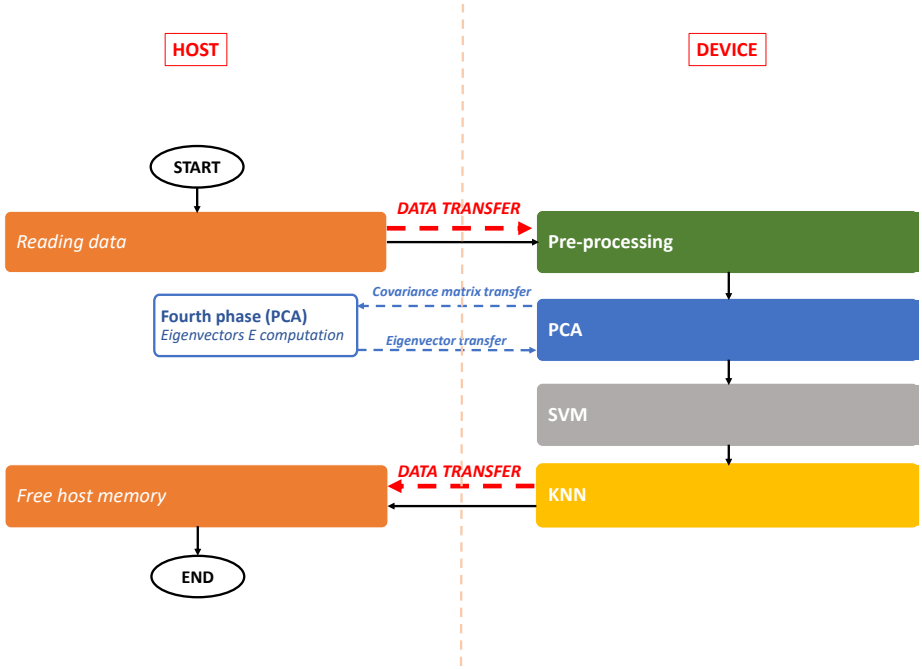


Figure 3.28: SSC – Version 1. First version of the Spatial-spectral supervised classification system. The pre-processing and all the algorithms are excuted in cascade on the GPU.

algorithms. All these data are transferred from the host to the device as the pre-processing phase starts. In the previous paragraphs, it has been explained that both in the pre-processing and in the PCA computation the data transfer has been performed using *streams*. In this case, *streams* are exploited only at the beginning of the flow, when the HS cube, the white and the dark images still have to be transferred for the calibration step. Once the pre-processing is completed, the data needed by the PCA are already stored in the GPU global memory. In this first Spatial-spectral classification system parallel implementation (which will be referred in the following as *SSC – Version 1*), the most performant version of each algorithm has been introduced in the flow. It must be noticed that the fastest version of each algorithm is the one executed on GPU. For this reason, most of the

execution flow is on the device, with the exception for the input reading, the declaration and the initialization of the variables as well as the memory release that takes place in the final part of the algorithm. Moreover, the host performs the eigenvectors computation since it is the fastest in providing this step result, as previously described in Paragraph 3.4.2. In this first solution, the SVM parallel code has been included even if the serial one is also well performing, as described in the Paragraph 3.4.3. This choise is justified by the fact that there is a unique handle declaration, which is used by different algorithms. In this case, the effective time of the parallel SVM computation is lower than the serial version. The code on device starts with the pre-processing step, whose output serves as input of both the PCA and the SVM, which are executed in cascade. Then, outputs are sent to the KNN which is the last step of the GPU flow. Finally, an array containing the pixels labels is transferred from device to host. The aim of the next three versions (referred in the following as *SSC – Version 2.1*, *SSC – Version 2.2* and *SSC – Version 2.3*) is to overlap some phases of the flow in an effort to save computational time. It must be noticed that the SVM and the PCA are two algorithms that can be computed simultaneously. Therefore, the idea is to perform the SVM on the host while some parts of the PCA are executed on the device. Figure 3.29 presents the flow of these three versions. The initial part of the flow is the same for all the versions: the input reading, the variable declaration and initialization are performed on the host. Then, as the data are transferred on the device, the computation begins with the pre-processing phase. Once the input matrix has been pre-processed, the PCA and the SVM can be performed. It is important to underline that CUDA allows to overlap a kernel execution on the device with a host function, if they are called consecutively [23]. In the case of Version 2.1, the SVM is the part performed on the host since it showed good performance in the serial execution. Therefore, the idea is to overlap the SVM with the covariance matrix computation (*Third Phase* of the PCA). In this version, the eigenvectors computation is on the device exploiting the suitable cuSOLVER function described in the paragraph 3.4.2 [68]. As the SVM and the PCA – *Third Phase* are executed, the SVM output is transferred on the device and the flows continues with the last PCA functions

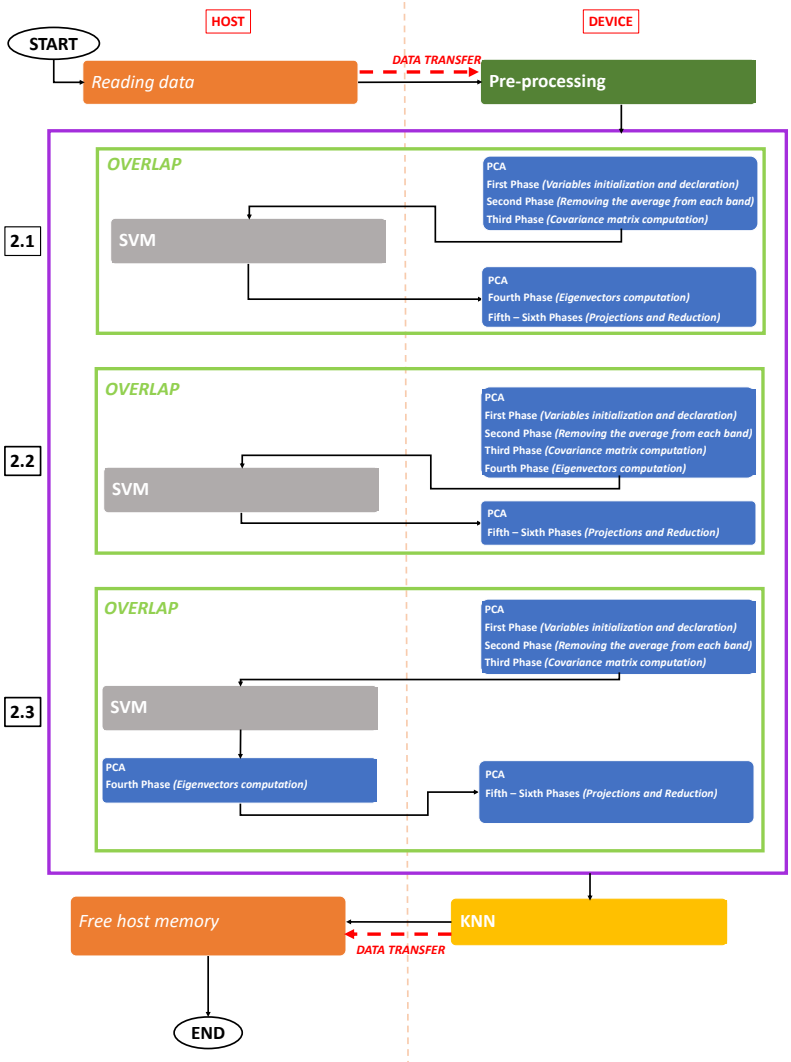


Figure 3.29: SSC – Version 2. These three versions differ from the parts of the flow that are overlapped. In the Version 2.1, the *Third Phase* of the PCA and the SVM are overlapped. In the Version 2.2, the idea is to execute the *Fourth Phase* of the PCA and the SVM simultaneously. In the last option, Version 2.3, the aim is to perform the SVM and the *Fourth Phase* of the PCA together with the *Third Phase* of the PCA.

and, then, with the KNN. Figure 3.30 is generated through the NVIDIA Visual Profiler and shows the overlap of these two parts: the green box surrounds the SVM bar which is performed on the host while the red boxes surround the PCA phases. It is possible to notice that the *Third Phase* of the PCA algorithm is executed on the device together with the SVM on the host. Considering again the Figure 3.29, instead of the Version 2.1, another option of the flow is to overlap the eigenvector computation, i.e. the *Fourth Phase* of the PCA, with the SVM (Version 2.2). Also in this case the eigenvectors are computed on the device while the SVM on the host. The idea is to evaluate if it is more convenient to overlap the SVM with the *Third Phase* (as in the case of the Version 2.1) or with the *Fourth Phase* of the PCA (Version 2.2). Analyzing the timeline generated by the Visual Profiler, in this case there is no overlap since the cuSOLVER function, exploited in the eigenvectors parallel computation, needs the host to call some intrinsic *cudamemcpy* functions, i.e. the ones that allow data transfer from host to device and viceversa. For this reason, Version 2.2 times will be not considered in the results analysis. The third developed option (Version 2.3) is the one that aims to overlap the SVM and the covariance matrix computation (*Third Phase* of the PCA), as already presented in Version 2.1. But, in this case, the eigenvectors computation is performed on the host, as in the best performing PCA version. At the end of the PCA and SVM computation, performed by one of these versions, the KNN can be executed. At the end, the result is transferred from device to host where the memory is released.

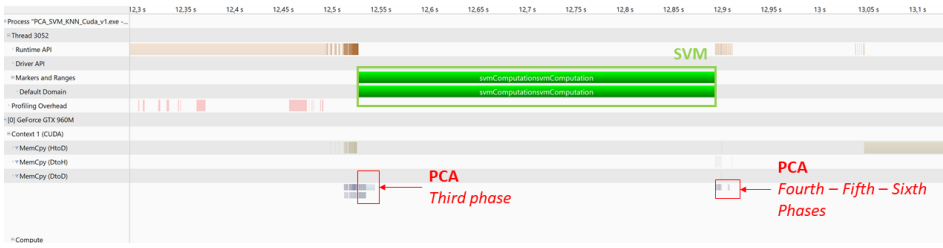


Figure 3.30: SSC – Version 2.1 timeline. This graph shows the timeline generated with the NVIDIA Visual Profiler.

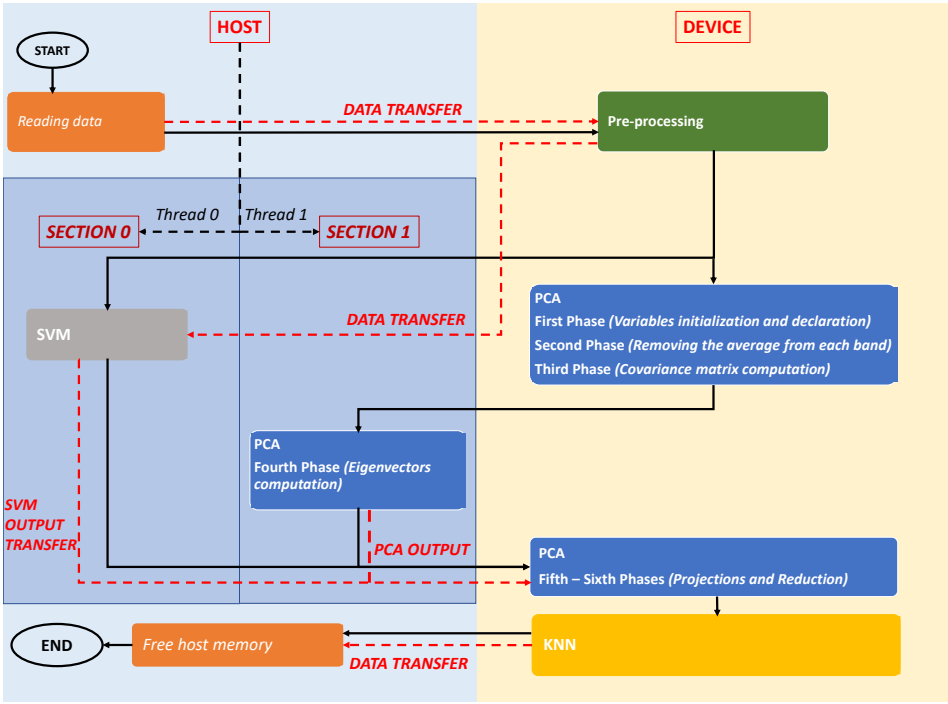


Figure 3.31: SSC – Version 2.4. After the pre-processing computation, two sections are generated in order to execute the SVM and the PCA in parallel.

As it is possible to see in Figure 3.29, in the Version 2.3 the SVM and the *Fourth Phase* of the PCA are sequentially executed.
In Version 2.4 a different approach has been introduced. Figure 3.31 shows that the flows starts again from the host with the input reading. Once read, the data are transferred to the device for the pre-processing computation. The pre-processed image is stored in the GPU memory but it is also transferred to the CPU. In this version two sections are generated and each one is managed by a CPU thread. The idea is that while the *thread 0* manages the serial SVM computation in *Section 0*, the *thread 1* manages the PCA computation in the *Section 1* which is performed both on device and on

host. In fact, the OpenMP API [15] allows to distribute the work among different threads, each one bounded to a parallel region called *section* [15]. The directive to generate these two parallel regions is described in the Algorithm 7.

```
1  #pragma omp parallel sections
2  {
3    #pragma omp section
4    {
5      In this section the thread 0 manages the SVM computation.
6    }
7    #pragma omp section
8    {
9      In this section the thread 1 manages the PCA computation.
10   }
11 }
```

Algorithm 7: Sections generation through the OpenMP directives.

After the SVM and the PCA ended, their outputs are sent to the KNN. At this point, the two sections end and, on the host, the computation is managed by only one thread (*master thread*). The flows continue with the KNN algorithm on the device. At the end, the result is transferred from device to host, as said before. The main difference between this version and the Version 2.3 is that in this case the SVM and the PCA are executed in parallel while, in the previuos code, only the *Third Phase* of the PCA and the SVM were performed simultaneously. The reason is that, in the previous case, the CPU thread had to manage the host/device transfers and viceversa, to execute the PCA both on CPU and on GPU, and moreover, it had to manage the SVM on the host.

The last developed version (Version 3) performs a double computation of the pre-processing phase, one on the host and the other on the device (Figure 3.32). Unlike the previous versions, in this case the idea is to duplicate the pre-processing step in order to have the pre-processed image both on host and on device, avoiding its transfer from device to host. As said in Paragraph 3.4.1, the pre-processing algorithm is made up of two main phases: the calibration and the normalization. This version has been developed so that the CPU-calibration and the CPU-normalization are overlapped with the GPU-calibration and GPU-normalization, respectively. Analyzing the

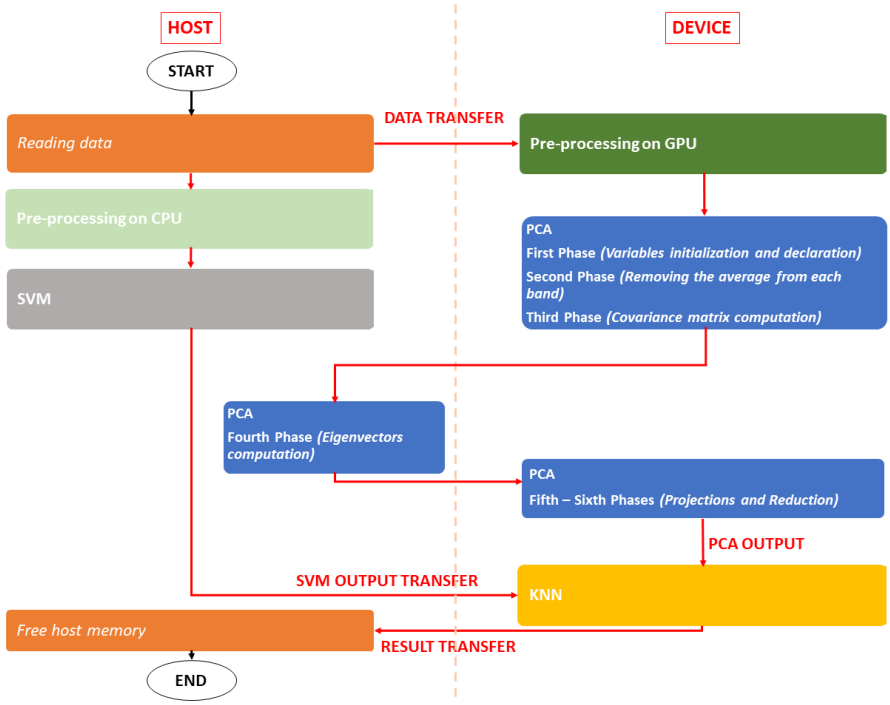


Figure 3.32: SSC – Version 3. In this version a double pre-processing computation is performed.

timeline generated by the NVIDIA Visual Profiler it is clear that only the two calibrations are overlapped since, during the normalization, the host is managing also other computation tasks that do not allow a simultaneous execution. Table 3.14 shows the computational times of all these versions performed on the *System 1* described before. The results presented in Table 3.14, show that the parallel versions allow to save time compared to the serial code. The serial Spatial-spectral system takes about five minutes to classify the biggest image of the database (P1C2) while the fastest parallel execution takes about 22 s. It must be noticed that all the parallel versions take almost the same time to classify the images and there is not a version that is significantly faster than the others. It is clear though that Version

			Computational time [s]				
Image ID	**Pixels**	**Serial**	**SSC-Version 1**	**SSC-Version 2.1**	**SSC-Version 2.3**	**SSC-Version 2.4**	**SSC-Version 3**
P1C1	251,532	281.48	21.27	21.46	22.42	21.40	23.98
P1C2	264,408	296.99	22.06	22.26	22.52	22.64	25.35
P2C1	219,232	222.67	17.48	18.09	17.74	17.61	19.95
P3C1	184,875	188.16	14.87	15.31	15.08	15.15	16.97
P4C1	124,033	85.17	7.08	7.26	7.06	7.05	8.33

Table 3.14: Results of the Spatial-spectral supervised classification system. The table shows the computational times of the SSC versions. The results correspond to the average of three executions. The times are expressed in seconds.

1 is the fastest one. This result highlights that, in this case, it is better to perform all the algorithms in cascade on the GPU rather than to increase the transfers number (from host to device and viceversa) in order to split the work between host and device. In this case, the simultaneous execution of algorithms (as the SVM and some phases of the PCA for the Versions 2.1-2.3 and the SVM and the PCA for the Version 2.4) is not convenient. As a matter of fact, performing the SVM on the GPU in cascade with the others algorithms proves to be a faster solution compared with the distribution of the algorithms between host and device due to the increased number of needed data transfers. Furthermore, it can be concluded that it is not convenient to duplicate the pre-processing phase (one performed on GPU and one on CPU) as done in Version 3, since the serial pre-processing computation takes more time than the pre-processed image transfer from the device to the host. The times presented in Table 3.14 refers to the system which contains the KNN *reference* version, i.e. the one characterized by *WSize14*. Table 3.15 presents a comparison between the serial code and the SSC-Version 1 (*SSC-Version 1 (WSize14)* in Table 3.15) results, both considering the *reference* KNN, and the SSC-Version 1 times obtained considering the KNN algorithm *best solution* characterized by *WSize8* and where the distances between pixels are computed exploiting the Euclidean metric, as described before. Also in this case it is possible to save computational time considering the *best* KNN solution of *reference* one. Both the parallel versions are faster than the serial code, with a speed-up that can be up to 22x.

		Computational time [s]			Speed-up	
Image ID	**Pixels**	**Serial**	**SSC-Version 1 WSize14**	**SSC-Version 1 WSize8**	**Serial/Version 1 - WSize14**	**Serial/Version 1 - WSize8**
P1C1	251,532	281.48	21.27	13.07	13.23x	21.54x
P1C2	264,408	296.99	22.06	13.89	13.46x	21.38x
P2C1	219,232	222.67	17.48	10.19	12.74x	21.85x
P3C1	184,875	188.16	14.87	8.92	12.65x	21.09x
P4C1	124,033	85.17	7.08	5.31	12.03x	16.04x

Table 3.15: Comparison between the Spatial-spectral classification versions characterized by different KNN algorithms. The table shows the computational times of two different Spatial-spectral classification system versions: the former is characterized the *reference* KNN algorithm, the latter considers the KNN version chosen previuosly as the *best solution* (*WSize8*). The serial code is characterized by the *reference* KNN version. Also the speed-up between these two parallel implementations and the serial code are shown. The times correspond to the average of three executions and are expressed in seconds.

3.4.6 K-means

The algorithm and the serial version

The K-means algorithm performs an unsupervised classification since a labelled dataset is not needed. The algorithm generates groups, called *clusters*, separating tissues and materials present in the hypercube, on the base of their spectral similarity. Each cluster centroid represents a spectra of a particular material present in the image. The number of clusters is one of the algorithm inputs. The serial K-means flow is shown in Figure 3.33. The inputs are the pre-processed image Y, the centroids number K_C, the threshold error (*min_error*) and the maximum iteration number (*max_iter*), used as stopping criteria. The *First phase* of the algorithm consists of the random initialization of the K_C centroids. Furthermore, an array called *previous_centroids* is initialized to 0. Finally, the iteration number *n_iter* is set to 0 and the *actual_error* is initialized to a huge value. The flow continues with a *while* loop. If the loop condition shown in Figure 3.33 is satisfied, the second phase can start. The distances between the pixels and the centroids are computed exploiting the Spectral Angle (SA) metric, as shown in Equation 3.17:

$$\theta(x,y) = \cos^{-1}\left(\frac{\sum_{h=1}^{M} x_h y_h}{(\sum_{h=1}^{M} x_h^2)^{1/2}(\sum_{h=1}^{M} y_h^2)^{1/2}}\right) \tag{3.17}$$

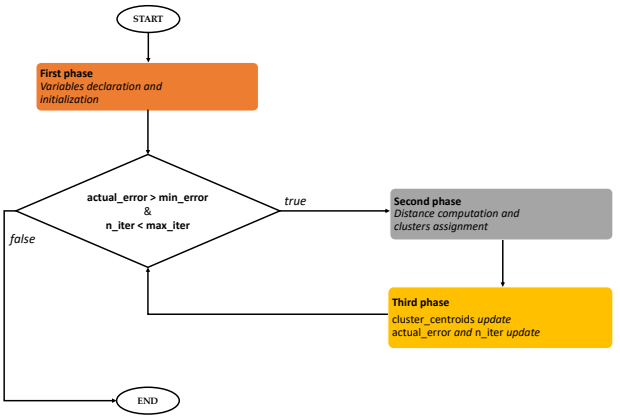


Figure 3.33: K-means serial flow.

where M is the number of bands, x and y are the spectral vectors and x_h and y_h represent the response of the h-th band of x and y respectively. The label assigned to each pixel corresponds to the cluster which has the centroid with the minimum value of SA. At this moment, the centroids used in the distance computation are stored in the *previous_centroids* array. The *Third phase* consists in the centroids update computing the barycenter of each group. The variation between the actual and the previous centroids is evaluated in order to update the *actual_error* variable. This variation represents how much the centroids changed and, for this reason, the *actual_error* is used as stopping criteria. At the end, the *n_iter* variable is updated to control the iterations number that the algorithm is performing. Unlike the other algorithms presented before, this classifier has not a fixed number of steps. In fact, the computation of new barycenters continues until the loop condition is false, that is when the difference between the centroids of two consecutive iterations is smaller than a fixed threshold (*min_error*) or when a maximum number of iterations is reached (*max_iter*) [48]. The output is a segmentation map showing different clusters characterized by several colours. In this system, this classifier is used in order to delineate the boundaries of the different spectral area present in the image which

should be identified by a specialist.

The parallel version

During a master thesis held in the Custom Computing Programmable System laboratory (University of Pavia), in collaboration with the IUMA (ULPGC), three CUDA versions of this algorithm have been developed. In Figure 3.34 the flow of the most performing is shown. The *First phase* is performed on the host and, then, the pre-processed image is transferred to the device. On the host the *while*-loop condition is evaluated and, if it is true, the array which contains the clusters centroids is transferred from host to device. Here, a custom kernel performs the distances computation. This kernel is characterized by a 2-dimensional grid whose rows refer to the pixels and the columns to the clusters. In this way, each thread computes the distances between a pixel and all the centroids. The distances

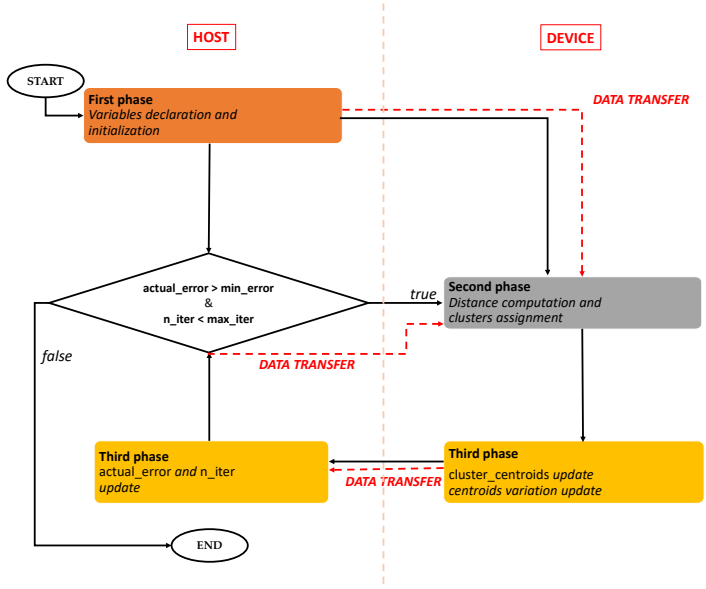


Figure 3.34: K-means parallel flow.

			Computational time [s]			speed-up	
Image ID	**Pixels**	**# of iterations**	**Serial**	**CUDA System 1**	**CUDA System 2**	**Serial/CUDA System 1**	**Serial/CUDA System 2**
P1C1	251,532	13	214.52	3.99	2.41	53.76x	89.01x
P1C2	264,408	25	465.61	7.45	3.16	62.48x	147.31x
P2C1	219,232	10	151.50	2.96	2.02	51.18x	75.00x
P3C1	184,875	13	162.37	2.97	1.94	54.67x	83.70x
P4C1	124,033	32	272.20	4.52	2.37	60.22x	114.85x

Table 3.16: Comparison between the K-means serial and CUDA versions. The table shows the computational times and the speed-up of the serial and parallel codes obtained exploiting the *System 1* (considering only one Tesla K40) and the *System 2*. Also the iterations number of each execution is presented. The computational times are expressed in seconds.

are stored in a $N \times K_C$ array where N is the pixels number and K_C is the clusters number. The second step of the *Second phase* is the *cluster assignment*: each thread of the kernel *findMinimum* searches the minimum distance between the pixel and a centroid and assigns to the pixel the label of the correspondent cluster. This kernel performs a pixel-wise computation. The cluster centroids update, in the *Third phase*, is computed by a simple kernel where the i-th thread manages the update of the i-th centroid. Once the update is completed, the variation between the previous and the actual centroids is evaluated exploiting the *cublasSasum* routine (CUBLAS library). This value is transferred on the host to compute the *actual_error* and the iteration number, *n_iter*, is incremented.

K-means results

The K-means serial and parallel versions have been tested exploiting both the *System 1* and the *System 2* described before. Table 3.16 shows the computational times and the obtained speed-up. An analysis has been conducted to estimate the value of the error, the maximum number of iterations and the number of clusters in order to have the best performance in the classification results and in the computational times. The number of clusters K_C is set to 24, the maximum number of iterations *max_iter* is 50 and the minimum error *min_err* is 10^{-3}. The presented results highlight that the parallel versions provide a significant speed-up. In the case of the biggest image of the database, P1C2, the CUDA version takes 7.45 s on

the *System 1* and only 3.16 s on the *System 2* instead of about 8 minutes (465.61 s) for the serial code. What it is important to notice is that this algorithm is faster if the *System 2*, equipped with the NVIDIA GTX1060 GPU, is used. Despite the GPU of the *System 1* (Tesla K40) optimized for scientific computations, this algorithm performs better on a more recent architecture whose CUDA cores work at a higher frequency than the Tesla ones. Finally, to graphically observe the classification results, Figure 3.35 shows the map generated by the K-means algorithm where each cluster is represented with a different colour. This kind of clustering allows to delineate with high accuracy the boundaries of some biological structures such as blood vassels, materials like the ring markers and different tissues. Even if the clusters have not an histological meaning, the region of interest is well delimited. This result is consistent with the main goal of the system that is to delineate the tumor area with high accuracy.

3.4.7 Majority Voting

The brain detection system presented in this work performs a hybrid classification: as explained in the previous paragraphs, it is characterized

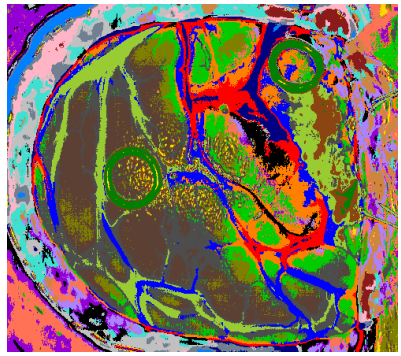


Figure 3.35: K-means classification map. This image shows the classification result of the image P2C1. The number of clusters present in the image is 24, each one represented by a color randomly selected.

by a supervised and unsupervised learning methods. The first one provides a classification map in which each pixel belongs to a class: this map is obtained exploiting the diagnosis information provided by the neurosurgeons and medical doctors. Even if this kind of output assigns a label to each pixel, it does not provide an accurate delineation of the tumor area. On the other hand, the unsupervised classification provides a good association of homogeneous areas in the image, even if the clusters that it generates are meaningless. These two approaches are computed independently but their results have to be merged in order to exploit their advantages and to obtain a final classification map. The Majority Voting is the approach selected to join these outputs, already used by authors of [71]. Figure 3.36 [48] shows an example of the Majority Voting technique. In this example, the unsupervised classification map delimitates the borders of each cluster area, but each cluster does not present a semantic meaning. On the other hand, the supervised map shows four different classes, each one represented by a specific colour and with a defined meaning. The two maps are merged to obtain the final hybrid result which shows each pixel belonging to a specific group, assigned on the base of the class of the supervised classification, also taking into account the clusters obtained by the unsupervised classification map (that delimitates the border of each cluster area) [48].

3.5 Parallel versions of the complete system

In the previous paragraphs the serial and the parallel versions of all the algorithms that belong to the brain cancer detection system have been presented. The aim of this paragraph is to explain how the *best solutions* of these algorithms have been merged exploiting the *single-GPU* and the *multi-GPU* architectures in order to obtain the most efficient version of the system. The goal is to reach a real-time classification that, in this particular case, is defined by a threshold of 1 minute, that is the time that the camera takes to capture the hyperspectral image. In the previous paragraph the complete serial system flow has been shown. The Figure 3.37 summarises the detection system serial flow (black lines), highlighting the inputs and

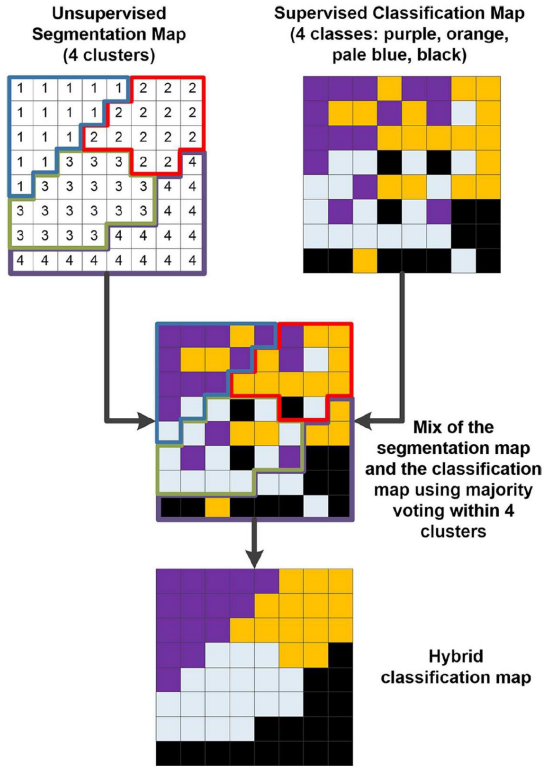


Figure 3.36: Majority Voting. [48]

the outputs paths (dotted lines) and graphically showing the classification output of each algorithm. What is represented in Figure 3.37 is the starting point for a parallel system development. The first parallel version designed exploits a *single-GPU* system and it is characterized by the execution of all the algorithms in cascade on the device (Figure 3.38). The flow starts on the host where the input image and the other variables are declared and initialized. There are two types of inputs: on one side there are the inputs

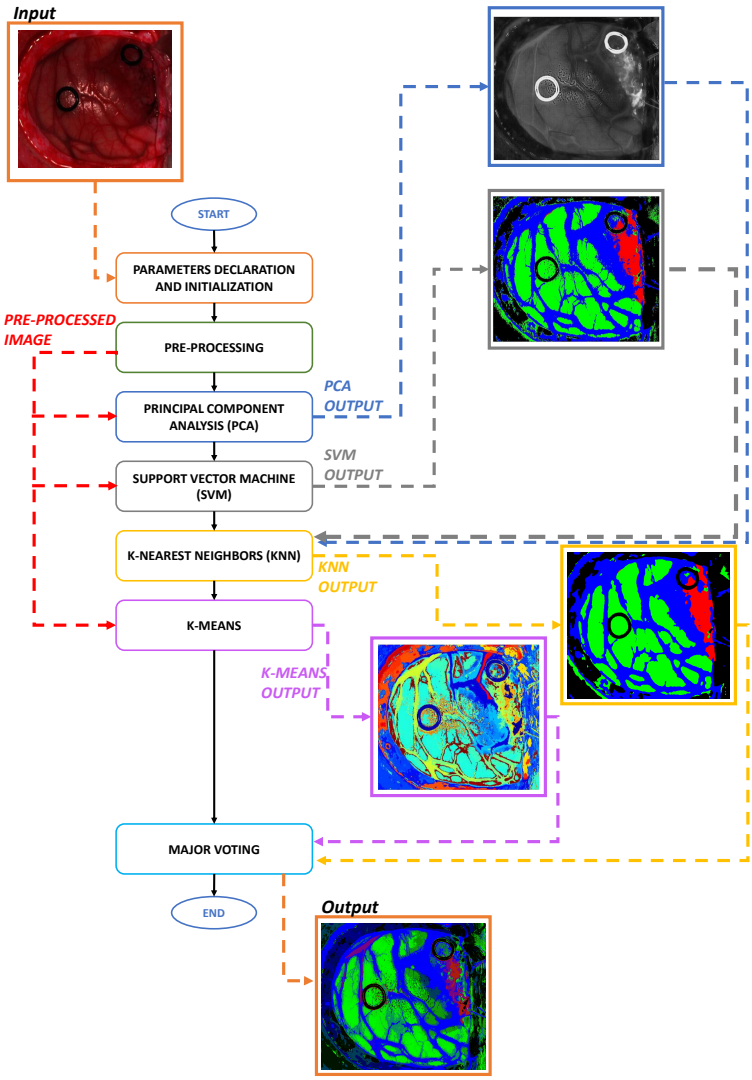


Figure 3.37: Serial flow of the brain cancer detection system. The image shows the flow of the serial code (black arrows), the input/output flow (dotted lines) and the graphic output of each algorithm.

read from the user interface. As it will be explained in the Paragraph 3.7, the user can set some parameters such as the number of neighbors K, the metric used in the distance computation, the *WSize* concerning the KNN algorithm, the maximum iterations number, the error and the number of clusters K_C concerning the K-means. The interface provides the *default* value of these variables (determined on the basis of the analysis presented before) but the user can choose to modify these values. The image and all these variables are read and stored in the CPU memory and transferred on the device for the computation. The other type of input is the one that is independent from the user: in fact, the white and dark calibration matrices and the variables belonging to the SVM model generated by the SVM training phase are also read, stored and transferred to the device. Once the data transfer is completed, the GPU parameters and variables are declared. For example, in this step the *handle* used in the cuBLAS routines and the *streams* used in the pre-processing are created. Furthermore, the grids and blocks dimension of the kernels are defined. Once that the GPU initialization is completed, the pre-processing can start. As explained in the Paragraph 3.4.1, the white and dark matrices and the input image are transferred exploiting CUDA *streams*. Once the image is calibrated and normalized (pre-processing phase), the pre-processed image is stored in the GPU memory and it is transferred to the host since it is used for the clusters initialization (K-means algorithm) performed on the CPU. The flow continues with the PCA algorithm which loads the pre-processed image from the GPU *global* memory. It is important to underline the difference with what explained in Paragraph 3.4.2. If the PCA algorithm is considered alone, the pre-processed image is an input read by the host and transferred on the device using *streams*. In the case of the complete system (and also in the Spatial-spectral classification system seen above), the pre-processed image is already on the GPU memory since the pre-processing step has been performed on the device. For this reason, considering the complete system, in the PCA there is no need to transfer the image. The PCA version included in this system is the one chosen as the *best solution* characterized by the eigenvectors computation on the host. The PCA output is the one-band representation of the hyperspectral-image which is stored in the global

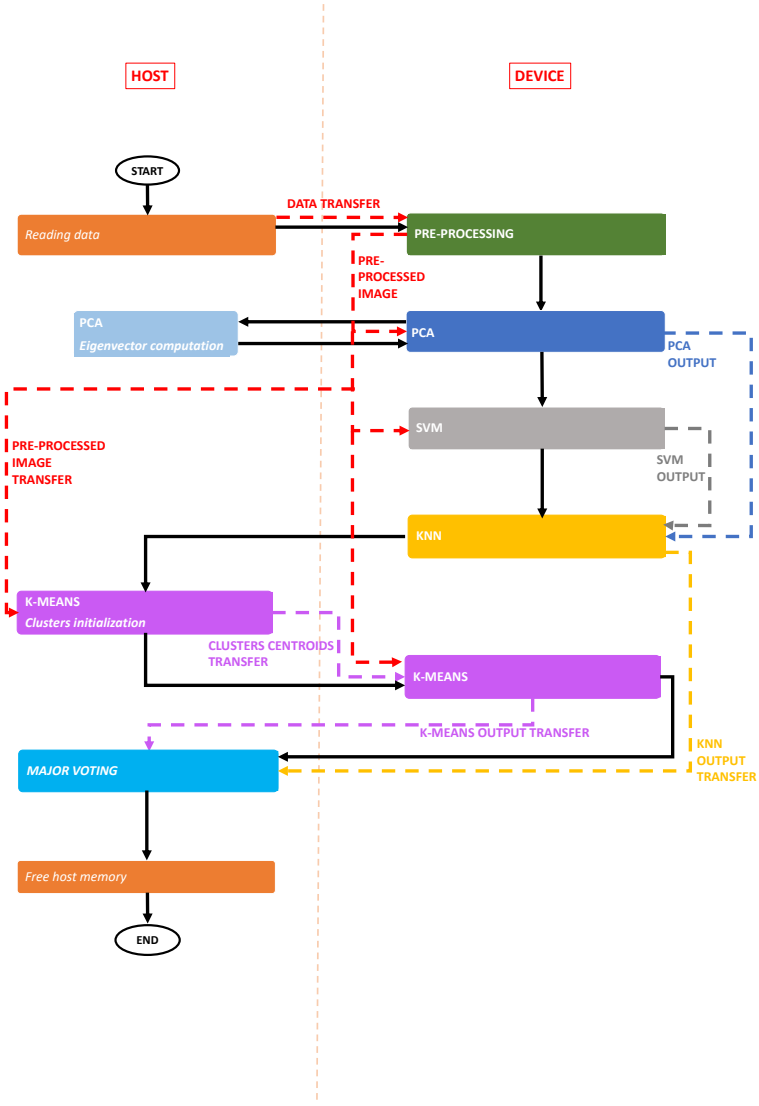


Figure 3.38: Complete system parallel version on *single-GPU* technology. First parallel version of the complete system. The image shows the flow of the parallel code (black arrows), the data transfers (dotted lines) host/device and viceversa and also between the algorithms.

memory in order to be used in the KNN algorithm. The pre-processed image is also a SVM algorithm input which is completely performed on the device. The probability map generated by the SVM classifier is also stored in the GPU global memory because it will be used in the filtering part of the KNN algorithm. The first kernel of the KNN algorithm needs the PCA one-band representation to compute the distances between the pixels in the neighbors selection phase. Once this step is concluded, the filtering kernel starts computing the optimized probability that a pixel has to belong to a class, taking into account the SVM probability map. The KNN output is an array containing the label of the class assigned to each pixel. The flow continues with the K-means algorithm, performed on device, whose inputs are the pre-processed image and the cluster centroids, which have been computed on the host during the *Clusters initialization phase*. Finally, the KNN and the K-means outputs are transferred to the host for the Majority Voting computation. The RGB image, i.e. the system output, is shown to the user through the user interface (Paragraph 3.7). As it will be presented in the next paragraph, this *single-GPU* version of the complete system classifies the hyperspectral image even in less than half a minute, satisfying the real-time constraint. To further reduce the computational time obtained with the *single-GPU* code, several *multi-GPU* versions have been developed. Analyzing the flows of the codes presented above, there are parts that could be executed simultaneously. For this reason, the idea is to perform these parts on different GPUs that work in parallel. In order to manage two boards in parallel, two *sections* (generated with the OpenMP API) are created: each one is characterized by a CPU-thread which manages the memory initialization, the variables declaration, the kernels launch and the transfers of each board. The flow of the first Complete System *multi-GPU* parallel version (*CS - multi1*) is shown in Figure 3.39. As in the previous version, the variables declaration and initialization is performed on the host. The inputs are transferred on one of the two boards (in this case the *Device 0*) where the pre-processing is performed. The pre-processed image is stored in the global memory of the GPU 0 and it is also transferred both on the host, for the *Clusters initialization* in the K-means, and on the other device (*Device 1*), for the K-means computation on GPU. This

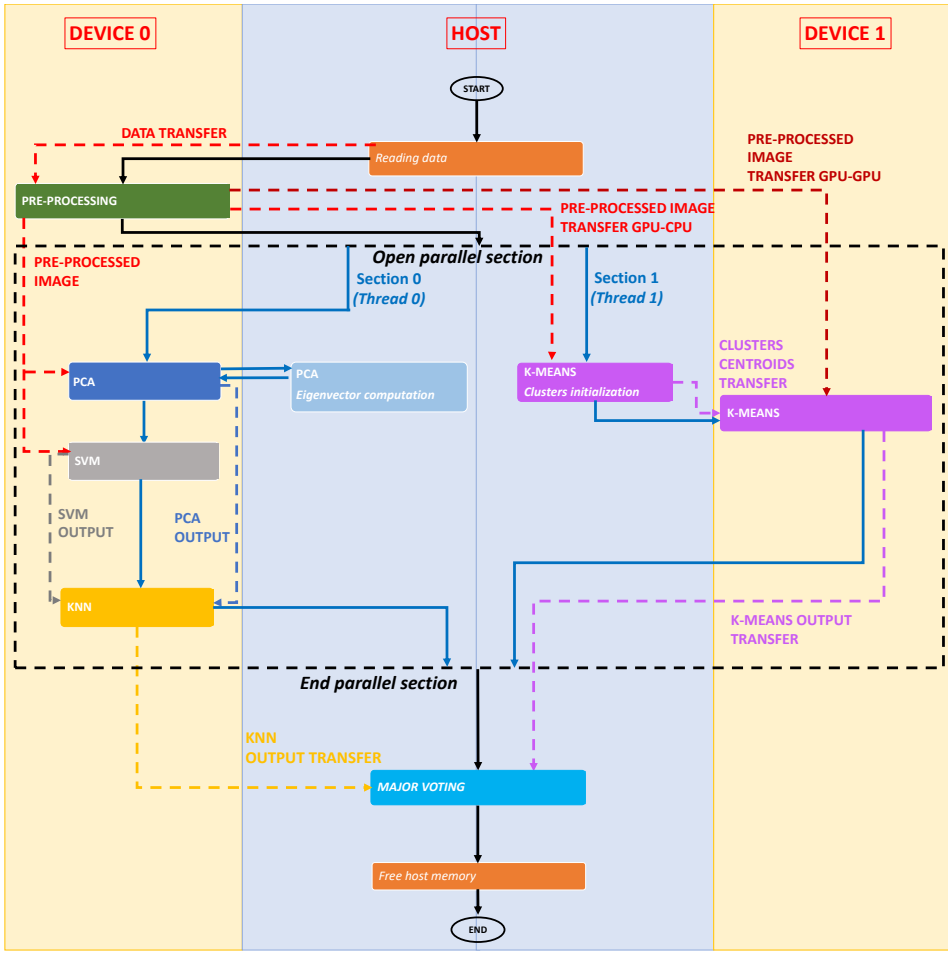


Figure 3.39: First complete system parallel version on *multi-GPU* technology (*CS - multi1*). The image shows the flow of the parallel code (black arrows), the data transfers (dotted lines) host/device and viceversa and also between the algorithms.

transfer between devices is possible thanks to the GPUDirect technology, as explained in Paragraph 1.2.2 in Chapter 1. In order to copy data from the memory of one device to the other, the first step consists in enabling the current GPU (in the case of the pre-processed image transfer is the GPU 0) to access addresses of the GPU 1. The function that allows this task is the *cudaDeviceEnablePeerAccess* whose input is the *destination device* index, that in this case is the GPU 1. As the access is enabled, the function that allows to copy data from Device 0 memory to Device 1 memory is the *cudaMemcpyPeerAsync*, whose parameters are:

- the destination device pointer *dev_preprocessedImage_gpu1*, i.e. the linear matrix in which the pre-processed image will be stored;
- the destination source that in this case is the GPU 1;
- the source device pointer *dev_preprocessedImage_gpu0*, i.e. the linear matrix that has to be copied on the GPU 1;
- the source device, i.e. the GPU 0;
- the size of the data that has to be copied that in this case is $N \times M \times dim_{float}$, where N is the number of pixels, M the number of bands and dim_{float} is the size of a float expressed in byte;
- the stream identifier.

Once the pre-processed image is transferred on the host and on the other device, a parallel section is opened exploiting the OpenMP directives shown in the Algorithm 8.

```
1  #pragma omp parallel sections
2  {
3    #pragma omp section
4    {
5      In this section the thread 0 manages the GPU 0 and performs the PCA,
     SVM and KNN.
6    }
7    #pragma omp section
8    {
9      In this section the thread 1 manages the GPU 1 and performs the K-means.
10   }
11 }
```

Algorithm 8: Sections generation through the OpenMP directives.

The execution is divided into two sections each one assigned to a different thread. As said before, each GPU is associated to a thread through the function *cudaSetDevice*. In this way the *thread 0* (Section 0) manages the *Device 0* and *thread 1* (Section 1) manages the *Device 1*. The CPU-threads work in parallel to compute simultaneously the different algorithms present in the two sections. On the one hand, the thread 0 manages the PCA, SVM and KNN computation in the same way described for the *single-GPU* code: the PCA is performed on the GPU 0, except for the eigenvector computation that is executed on the host. Also the SVM is performed on the GPU 0 and its output, together with the PCA output, is sent to the KNN, which is also executed on the device. The KNN output is transferred on the host through a *memcpy* function managed by the thread 0. On the other hand, the thread 1 computes the K-means *Clusters initialization* on the host, then it transfers the clusters centroids on the GPU 1 where the other steps of the K-means are computed. At the end, the output is transferred on the host. At this point the parallel section is closed and the computation is managed only by one thread. The Majority Voting is performed on the host and the new classified image is plotted. Another *multi-GPU* system version (*CS - multi2*) has been developed. It differs from the (*CS - multi1*) from the point where the parallel section is generated. As it is possible to see in Figure 3.40, the *Reading data* step is always performed on the host. After this phase, the parallel section is created in the same way described in Algorithm 8. The main difference is that, since the pre-processing computation

is performed in both the devices, each GPU requires the initial data. For this reason there is a double-transfer of the initial data, one managed by the thread 0 for the Section 0, the other by the thread 1 for the Section 1. The code in each Section can be executed in parallel: each thread manages the data transfer to its device where the pre-processing phase can start. After this step, in Section 0, the PCA, SVM and KNN are computed as explained before. In Section 1, the pre-processed image is transferred from the Device 1 to the host in order to compute the first part of the K-means. Then the computation continues on the device as described above. In this case there is no need of a GPU-GPU transfer since each device computes a pre-processing phase, so the pre-processed image is already stored in the global memory of each GPU. These last two versions performed on the *multi-GPU* system allow to further reduce the computational times of the brain cancer detection system as it is analyzed in the following paragraph.

3.6 Results

The development of an optimized version of the brain cancer detection system has required several steps and analysis before reaching a very efficient and performing version capable of satisfying the real-time constraint that, in this work, is set to one minute. As described in the previous paragraphs, the first step of the work has been the development of the optimized versions of the algorithms that characterized the HELICoiD system. Several strategies and implementations have been described above but, according to the results shown before, it is possible to conclude that the GPU technology together with an efficient design of the algorithm codes allows to reach high speed-up and to save computational time. Right after the choice of the best solution for each algorithm, it has been studied the most efficient way to link all these versions together in order to develop a highly optimized brain cancer detection system, exploiting both the *single* and the *multi-GPU* technologies. Table 3.17 shows the comparison between the computational times of the three complete system parallel versions and the serial one, presenting also the obtaining speed-up.

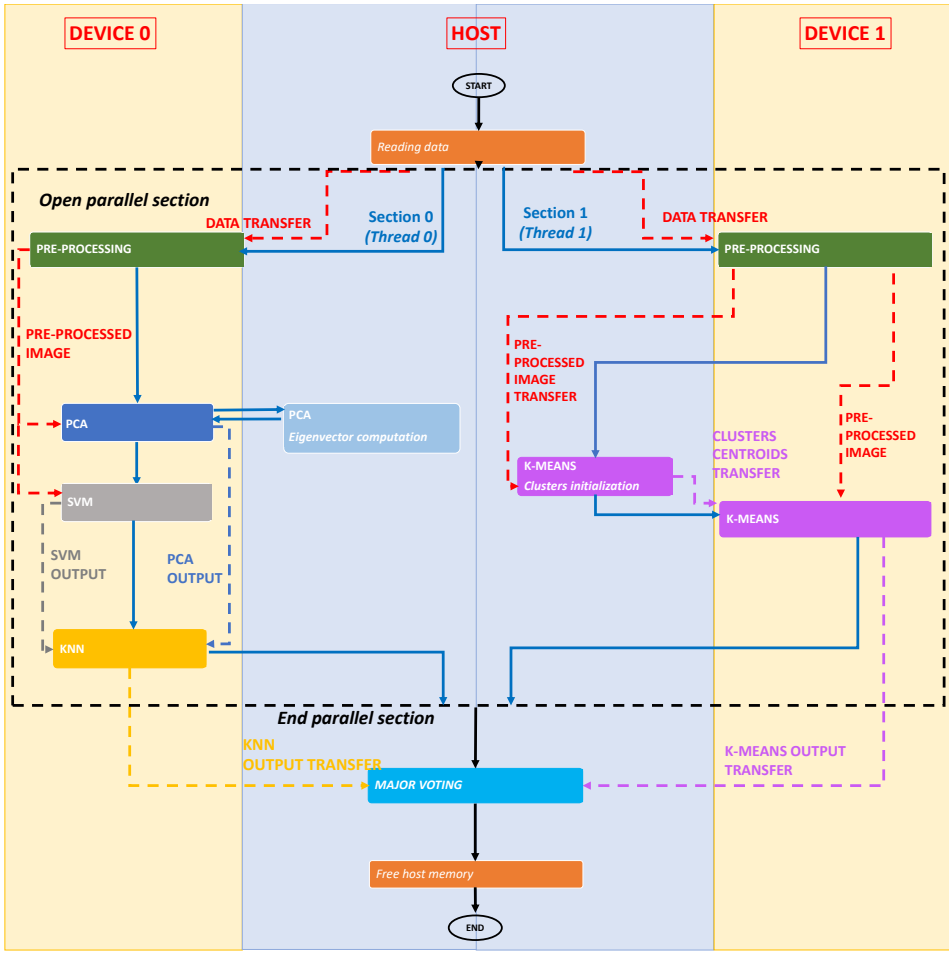


Figure 3.40: Second complete system parallel version on *multi-GPU* technology (*CS - multi2*). The image shows the flow of the parallel code (black arrows), the data transfers (dotted lines) host/device and viceversa and also between the algorithms.

			Computational time [s]			Speed-up		
Image ID	**Pixels**	**Serial version**	**CS - single**	**CS - multi1**	**CS - multi2**	**Serial/CS-single**	**Serial/CS-multi1**	**Serial/CS-multi2**
P1C1	251,532	600.22	25.66	21.26	21.38	23.39x	28.23x	28.07x
P1C2	264,408	719.31	27.66	22.53	22.38	26.00x	31.92x	32.13x
P2C1	219,232	459.74	20.99	17.66	17.86	21.90x	26.03x	25.74x
P3C1	184,875	387.43	18.64	15.67	15.17	20.78x	25.54x	25.53x
P4C1	124,033	301.17	9.37	7.27	7.17	32.09x	41.41x	42.03x

Table 3.17: Comparison between the serial and the parallel versions of the complete system. The table shows the computational times and the speed-up of the serial and parallel codes obtained exploiting the *System 1*, considering only one Tesla K40 in the *CS- single* code and both the boards in the *CS - multi1* and *CS - multi2* versions. The computational times are expressed in seconds.

An analysis of the results shown in Table 3.17 reveals that all the parallel versions classify the hyperspectral image in less than one minute, meeting the real-time constraint. Despite the *single-GPU* version allows a significant reduction of the computational times of the serial code, the fastest parallel solutions are the one that exploit the multi-GPU technology. It is possible to notice that the *multi-GPU* versions allow to save up to about 5 s compared to the code which exploits only one GPU. The reason is that the K-means can be performed together with the other algorithms, handing the two computations to two different GPUs simultaneously. If the biggest image of the database (P1C2) is considered, the *multi-GPU* versions take respectively 22.53 s and 22.38 s for the image classification (obtaining speed-up equal to 31.92× and 32.13× respectively), compared to the 27.66 s of the *single-GPU* version and the 719.31 s (about 12 minutes) of the serial code. The highest speed-ups are obtained with the P4C1 image classification reaching also the 42.03×. As it can be noticed from the Table 3.17, the two *multi-GPU* versions results are not very different. This means that, for this type of system, performing two pre-processing steps (one in each GPU) takes almost the same time than computing only one pre-processing and transferring the pre-processed image through a GPU-GPU data transfer. In fact, as it will be possible to see later, the data transfers are not so heavy in the execution. The results shown in Table 3.17 refer to those codes that include the KNN algorithm *reference* version, which performs the neighbors selection in a *searching window* characterized by 14 rows (*WSize14*) and exploits the Euclidean metric in the distance computations. In the Paragraph

			Computational time [s]			speed-up		
Image ID	**Pixels**	**Serial version**	**CS - single**	**CS - multi1**	**CS - multi2**	**Serial/CS-single**	**Serial/CS-multi1**	**Serial/CS-multi2**
P1C1	251,532	600.22	17.95	13.45	13.54	33.43x	44.63x	44.33x
P1C2	264,408	719.31	19.49	14.22	14.23	36.91x	50.58x	50.54x
P2C1	219,232	459.74	14.01	11.34	10.53	32.81x	40.55x	43.66x
P3C1	184,875	387.43	12.02	8.99	9.57	32.23x	43.19x	40.46x
P4C1	124,033	301.17	7.79	6.41	6.40	38.66x	46.98x	47.09x

Table 3.18: Comparison between the serial code and the parallel versions of the complete system, characterized by the KNN with *WSize8*. The table shows the computational times and the speed-up of the serial and parallel codes obtained exploiting the *System 1*, considering only one Tesla K40 in the *CS- single* code and both the boards in the *CS - multi1* and *CS - multi2* versions. The serial version presents the KNN *reference* version while the parallel codes present the KNN characterized by *WSize8*. The computational times are expressed in seconds.

3.4.4 we concluded that the version *WSize8*, whose searching window has 8 rows, and that computes the distance using the Euclidean metric, is a very efficient option to be considered for its computational time and classification results. For this reason, Table 3.18 shows the *reference* serial version compared with the parallel codes characterized by the KNN algorithm with *WSize8*. The KNN version with *WSize8* has provided a further reduction of the image classification computational times. With these new versions it is possible to classify the biggest image of the database (P1C2) in only 14.22 s if the fastest parallel version is considered. Concerning the *Single-GPU* version, the highest speed-up is obtained in the P4C1 image evaluation and it is 38.66×. With regards to the *multi-GPU* versions, the highest speed-up refers to the P1C2 image classification reaching the 50.58×.

The graphs A) and B) (Figure 3.41) show the timelines of the *CS- multi1* code considering the KNN - *WSize14* (A) and the KNN - *WSize8* (B), both exploiting the Euclidean metric in the distance computations. These graphs are the representation of the data generated by the NVIDIA Visual Profiler. Each bar represents a task that can be a transfer or a function performed by thread 0 (red), thread 1 (green), GPU 0 (orange) and GPU 1 (light blue). The longest task which is performed by the thread 0 is the image upload, that is the hyperspectral image reading, while it is important to highlight that all the data transfers (*cudamemcpy*) are very fast and sometimes negligible. This aspect underlines that the data have been prepared in an efficient way for the transfer so that the most of the execution

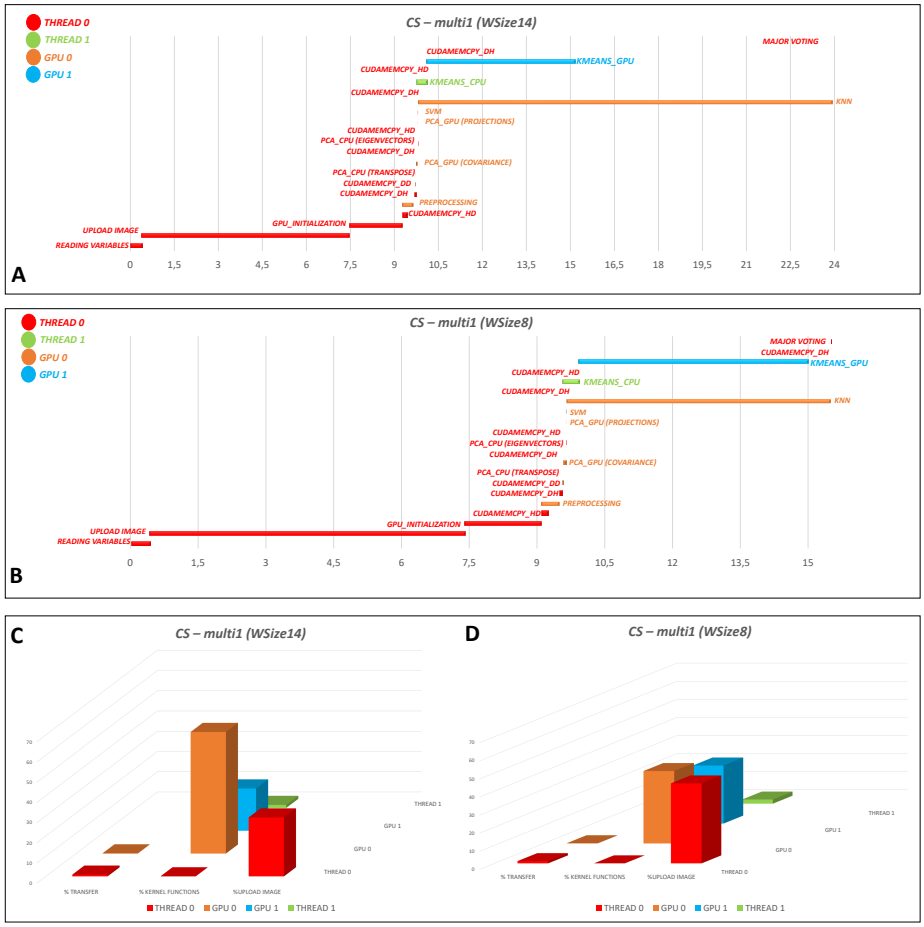


Figure 3.41: Timelines of the *CS-multi1* version considering both *WSize14* and *WSize8* KNN algorithms. A) Timeline of the *CS-multi1* version considering the KNN - *WSize14*. B) Timeline of the *CS-multi1* version considering the KNN - *WSize8*. C-D) Graphs that indicate the percentages of the transfers, of the functions and kernels and of the image upload for each thread and GPU, considering the *CS-multi1* version with the KNN - *WSize14* (C) and the KNN - *WSize8* (D).

is dedicated to the algorithms computation. Concerning the algorithms, it has been explained previously how their computation is managed by different threads or GPUs simultaneously. This is the reason why, analyzing the graphs, it can be noticed that, after the pre-processing computation, where a parallel section was generated, the red and orange bars are overlapped to the green and light blue ones. Considering both the graphs A) and B), it is interesting to observe how the result changes if the KNN with different *WSize* is considered. Focusing only on the KNN orange bar and on the K-means light blue bar, it is possible to notice that in the version with the KNN - *WSize14* (A) is the KNN that prevails on the computational time of the entire execution. But, if the window size is reduced (or if the Manhattan metric is chosen), also the KNN computational time decreases and the K-means becomes more and more significant in the computational time of the entire system. The graphs C) and D) (Figure 3.41), show the time percentages dedicated to the transfers, the kernel and functions execution and the image upload that each GPU thread has to manage. The upload image percentage is related only to the thread 0 (the highest red columns in graphs C) and D)) and in the case of the KNN - *WSize14* is 29.22% while in the case of the KNN - *WSize8* it reaches the 44.25%. This is a very significant data if it is considered that, in this last case, almost half execution is dedicated to the input reading instead of the computation. It is important to underline, though, that it is a temporary condition since the image is read from a file. The data will be transferred directly from the camera when it will be connected to the system. Considering the other tasks performed by the thread 0, it can be noticed that the percentages related to the transfers and the functions are very low: for example, the percentage of all the transfers performed by the thread 0 is 0.85% in C) and 1.24% in D). The thread 0 manages the GPU 0 (orange columns) which dedicates the most of the execution to the computation of the kernels and, in particular, to the KNN algorithm. Also in this case the transfers percentages are almost zero and the data related to the image upload is not present since it is performed by the thread 0. The thread 1 and the GPU 1 perform the K-means computation and, also in this cases (green and light blue columns), the most relevant data are the one related to the kernels

and functions computations. The main difference between the C) and D) graphs is the orange column which refers to the kernels computation. Passing from the KNN - *WSize14* to the KNN - *WSize8*, the duration of all the tasks remains almost the same (as it can be seen in graphs A) and B)) but the time of KNN algorithm, and so its percentage on the total execution, decreases. In graph D) (as explained for the graph B)) it can be noticed that the percentages of the kernels performed by the GPUs (orange and light blue columns) present more similar values than the previous case. In particular, in the graph C) the time percentage dedicated to the kernels computation by the GPU 0 is 60.16%, while the one dedicated to the kernels by the GPU 1 is 20.93%. In the graph D) these percentages are 40.08% and 32.13% respectively.
After that the computational results have been discussed, it is important to graphically analyze the classification results. As described in Paragraph 3.4.7, the last step of the complete system was the combination of the clustering map, generated by the K-means algorithm, with the supervised classification map, obtained with the KNN algorithm. On one side there is a map whose clusters have not histological meaning but they show well delineated borders; on the other side there is a classified image whose pixels are shown with four colors which represent a specific tissue (or background). The Majority Voting has combined these two different maps, with different information, in order to improve the final classification result. In Figure 3.42 two different ways to represent the final classification map are shown. In particular, it presents the comparison between the P2C1 classification maps obtained by the *CS- multi1* version considering both the KNN - *WSize14* (images A and B) and the KNN - *WSize8* (images C and D). Two methods have been exploited to represent the brain cancer detection system output. Figure A) and C) are generated considering the maximum majority class of the supervised classification map to each cluster of the K-means map. For example, if the maximum class present in a cluster is the tumor, the cluster RGB color will be R=1, G=0, B=0 (values expressed in percentage). These two maps provide a more accurate result compared to the map generated by the KNN algorithm because the borders of each class, and so of each tissue, are better delineated. Despite this, it should be noticed

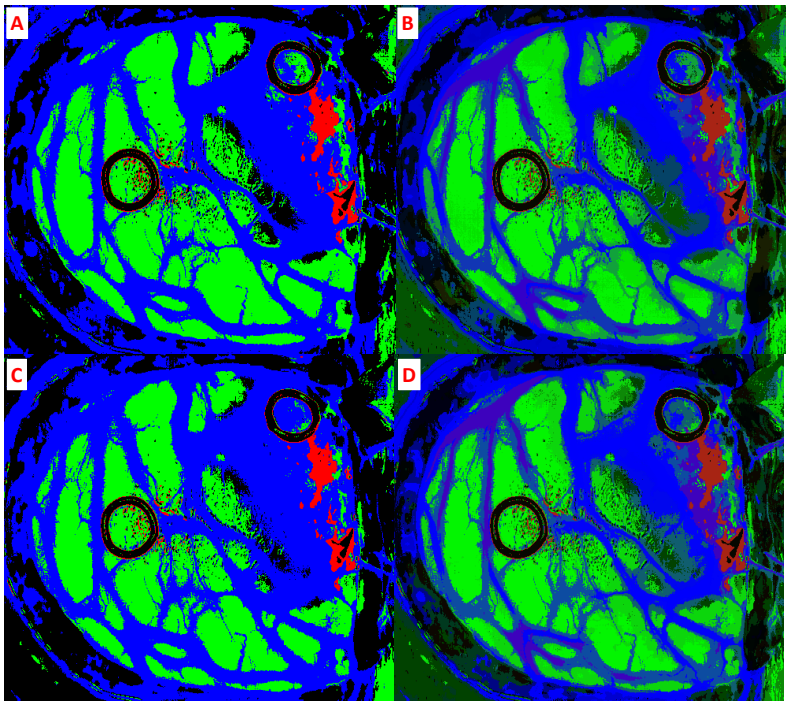


Figure 3.42: Classification result of the image P2C1. A) - B) Classification maps obtained with the *CS- multi1* version considering the KNN - *WSize14* with and without different tonalities. C) - D) Classification maps obtained with the *CS- multi1* version considering the KNN - *WSize8* with and without different tonalities. The red colour identifies the tumor area, the green the healthy tissue, the blue the hypervascularized tissue and the black the background.

that if in a cluster there is a high presence of another class (which is not the maximum class) the information related to this new class is not shown in these maps. For this reason another way to represent the classification map has been adopted. It is called Three Maximum Density (TMD) and it consists of mixing the colors of the classes present into a cluster taking into account the three maximum MV probability values [48]. For example, if the probability to find the hypervascularized tissue in a cluster is 70%, the

probability of tumor is 20% and of healthy tissue is 10%, the class assigned to the cluster is the hypervascularized one but the RGB color cluster will be R=0.2, G=0.1, B=0.7. Figure 3.42 B) and D) shows the classification maps generated with the TMD method. This technique allows to show different tissues present in a cluster providing more information than the maps generated taking into account only the cluster maximum class. The images A) and B) show the classification map (without and with different tonalities) generated by the *CS- multi1* version considering the KNN - *WSize14*. In Paragraph 17, it has been explained that the KNN classification results are the same if the entire image or a window with 14 rows are used as searching window in the neighbors selection. At this point it can be concluded that also the classification results of the *CS- multi1* version with the KNN - *WSize14* are the same than the ones of the original HELICoiD system which considered the entire image as searching window, and whose classification maps are validated by specialists and neurosurgeons. The images C) and D) refer to the *CS- multi1* version considering the KNN with *WSize8* and the Euclidean metric in the distance computation. In Paragraph 17, it has been explained that the classification difference percentage of the *WSize8* version compared to the reference one is of 0.029%. In fact, analyzing the KNN filtered maps (Figure 3.25) of these two versions, very few differences can be appreciated by the naked-eye. Also considering the comparison between the maps A) - C), and between the maps B) - D) (Figure 3.42) few differences can be noticed, especially if the tumor area is considered. For this reason, it can be concluded that the *multi-GPU* version of the complete system which considers the KNN algorithm with *WSize8* and the Euclidean metric, is a very good solution for the hyperspectral image classification since it provides an accurate classification map in real-time.

3.7 User Interface

Part of the work of another master thesis developed in the ULPGC in collaboration with the Custom Computing Programmable System laboratory (University of Pavia) consisted in the development of a user interface

of the brain cancer detection system. The aim of this paragraph is to briefly illustrate how the system is presented to the neurosurgeons in order to give a more complete description. Figure 3.43 shows the main windows of the interface. Figure 3.43.A presents the main window of the program: in the top-left side there are three buttons (to load the image, to change the parameters and to start the classification) and a drop-down list, which shows the types of computation that the user can choose on the base of the technology that he wants to exploit. In this case (Figure 3.43.B, yellow arrow) the user can select a serial computation performed by a CPU, a multi-core execution or a parallel computation performed by a single or a multi-GPU technology. The green arrow indicates the *parameters* button. If it is pressed, the window shown in Figure 3.43.C is opened. This window presents all the parameters of the different algorithms described above with the *default* values. The user can choose to change these parameters and try new system configurations. Once the new system configuration has been chosen, the user can upload the image (Figure 3.43.D) and press the button *Process* to start the computation. At the end, the final result will be shown as in Figure 3.43.F.

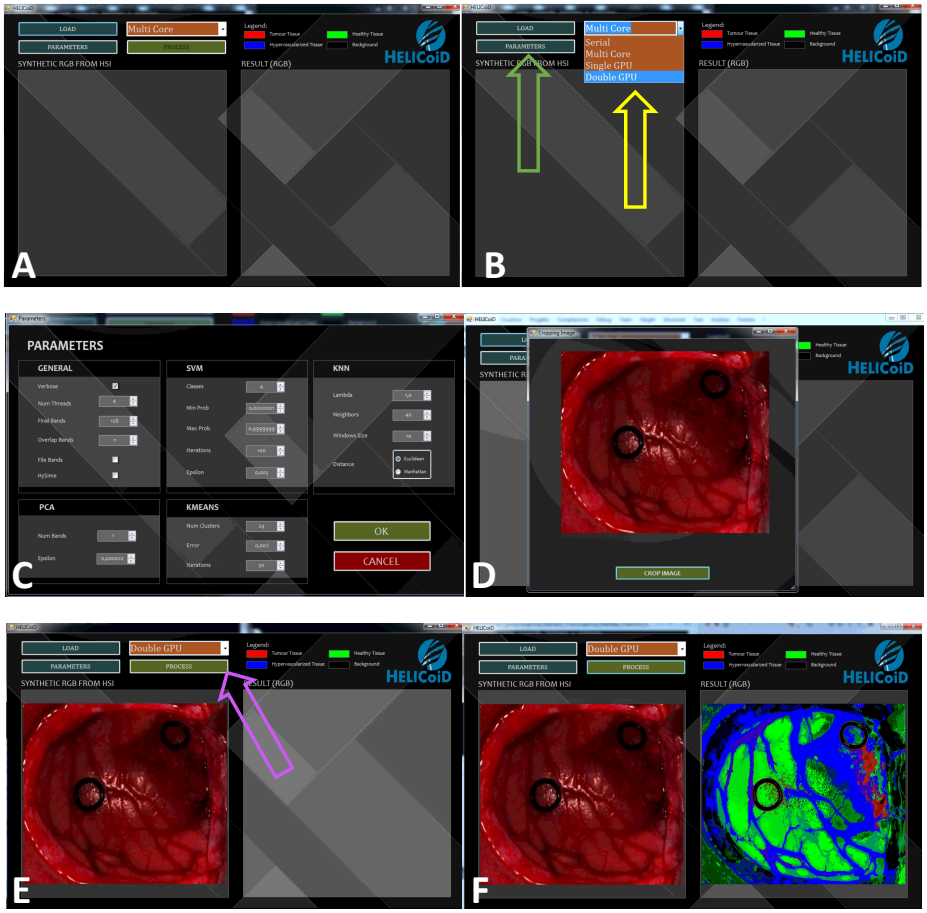


Figure 3.43: User interface. A) Main window. B) Selection of the computation type and the parameters. C) Parameters window. D) Image upload. E) Window before the image processing. F) Final window with the classification map.

Chapter 4

Overall conclusions

Several aspects in the *personalized* medicine of the future require very powerful technologies able to elaborate huge amount of data in less time as possible or even in real-time. The work described in this thesis demonstrates the crucial role that HPC technologies have in two medical cases of study: the computational modelling and the support system development. The design of the cerebellar granular layer simulator belongs to the first research field. This work is part of the european Human Brain Project which aims at performing a complete simulation of the human brain. This task will bring several benefits such as a deeper brain knowledge which will allow to develop specific diseases diagnosis, personalized treatments and discover new pathologies and non-invasive therapies. A wide brain understanding will also provide a novel architecture philosophy for hardware devices: scientists develop innovative neuromorphic and neurorobotic systems based on neurons activity and brain elaboration capabilities. These systems will use the same basic principles of computation and the same cognitive architecture as the brain does. The University of Pavia is focused on the cerebellum developing complex mathematical models capable of reproducing its functionalities and behaviors in a realistic way. Since these models are heavy from the computational point of view, the use of

HPC technologies is crucial to reduce the elaboration times, aiming at the real-time simulation. During this work, several technologies and paradigms have been evaluated concluding that GPU is the best performing for this application. This work is focused on the development of a realistic granular layer simulator, which is part of the cerebellar cortex. The first phase of the work dealt with the reproduction of the Golgi and granule cells activities, developing realistic simulators of these cells exploiting the single- and multi-GPU systems. This technology allowed to significantly reduce the serial elaboration time. For example, the reproduction of 3 seconds of neuronal activity of 400,000 Golgi cells takes about 18 hours considering the serial algorithm. This time is reduced to about 28 minutes exploiting one GPU board and reaches about 15 minutes and only 313.10 seconds if multi-GPU systems are considered.
The Golgi and granule cells simulators are able to correctly reproduce the neuronal behaviors. During the second phase of the work the granular layer network has been developed following two main steps: the network design and its simulation. The former concerned the computation of the elements to insert in a 3D space, their displacement and their connection following detailed and realistic rules. The latter dealt with the reproduction of the network activity exploiting the Golgi and granular cell simulators. The algorithm which reproduces the network behavior is developed to exploit the GPU device. Also in this case, this technology has provided a faster result compared to the serial version. The activity (3 seconds) of a network made up of 194,333 granules, 972 Golgi, 32,399 glomeruli and 1,080 mossy fibers takes about 35 hours considering the serial code and 7.5 hours considering the parallel solution, using one GPU board. Despite this result is very significant, the first future work will be the network simulation on multi-GPU systems. Moreover, other cerebellar behaviors and functionalities have to be reproduced and validated. Finally, the Purkinje and molecular layers algorithms will be developed to have a complete cerebellar cortex simulator. The results obtained in this work are very important because they demonstrate how GPU technology can significantly reduce the elaboration time in this kind of applications, where complex models have to be solved as fast as possible. It is also true that the project goal is the real-time simu-

lation that will probably be achieved only using powerful supercomputers and manycore solutions, due to the complex mathematical task. Therefore, this thesis has demonstrated that the parallelization approach is the right way to follow to satisfy the real-time constraint.

The second case of study concerned the use of the HPC technologies in the development of a real-time version of a brain cancer detection system. This work, part of the HELICoiD project, has been carried out at the IUMA laboratory of the Universidad de Las Palmas de Gran Canaria, which is the project leader. This system acquires an hyperspectral image of the patient's brain and classifies it in order to display in real-time a map where the tumor parts are distinguished from the healthy tissue. During a neurosurgery, the doctor has to resect the cancer from the brain with an high level of precision since over-resection can cause important demages and, on the other side, leaving tumor tissue in the brain can cause recurrence. For these reasons, the system requires an high accuracy in the cancer delineation and a real-time classification to provide a fast help to the surgeon. Once acquired, the hyperspectral image is pre-processed and sent to the classification system which is made up of two main branches concerning the supervised and unsupervised classification. In the former the PCA and the SVM are performed and their outputs are sent to the KNN which provides the result of the supervised branch. The unsupervised classification dealt with the K-Means. At the end, the Major Voting technique joins the two results to develop the final classification map. The work described in this thesis focused on the development of a parallel version of this system in order to exploit the GPU technology to reach the real-time elaboration. In this specific application the real-time constraint has been set to 1 minute. The best parallel version of the algorithm is able to classify the biggest image of the database in about 14 seconds. This result demonstrates that the goal has been achieved but, in the future, this elaboration time has to be further reduced in order to classify hyperspectral video in real-time. For this reason, new techniques to optimized the code will be developed in order to exploit multi-GPU systems that, as shown in this work, are able to provide fast results.

These two cases of study demonstrated that HPC technologies and, in particular, GPUs play a crucial role in medical applications, where a great amount of data has to be elaborated as fast as possible. Nowadays, the healthcare and medical research require increasingly data to perform a more precise and personalized medicine for a specific patient. And in this context, HPC technologies will be essential.

Bibliography

[1] "HPC Simulation is Transforming the Future of Precision Medicine." https://www.hpcwire.com/solution_content/hpe/manufacturing-engineering/hpc-simulation-transforming-future-precision-medicine/.

[2] "The living heart project." https://www.3ds.com/products-services/simulia/solutions/life-sciences/the-living-heart-project/.

[3] "Human Brain Project." https://www.humanbrainproject.eu/en/.

[4] "Sarah Scoles: Will a digital twin save your life?." https://www.dallasnews.com/opinion/commentary/2016/03/04/sarah-scoles-will-a-digital-twin-save-your-life.

[5] "HELICoiD project." https://cordis.europa.eu/project/rcn/111274_it.html.

[6] H. Wang, H. Peng, Y. Chang, and D. Liang, "A survey of GPU-based acceleration techniques in MRI reconstructions," *Quantitative imaging in medicine and surgery*, vol. 8, no. 2, p. 196, 2018.

[7] S. S. Stone, J. P. Haldar, S. C. Tsao, W.-m. W. Hwu, Z.-P. Liang, and B. P. Sutton, "Accelerating advanced MRI reconstructions on GPUs,"

in *Proceedings of the 5th conference on Computing frontiers*, (Ischia, Italy), pp. 261–272, ACM, 2008.

[8] X. Sun, S. Wang, J. Song, L. Zhou, Y. Peng, M. Ding, and M. Yuchi, "Toward parallel optimal computation of ultrasound computed tomography using GPU," in *Medical Imaging 2018: Ultrasonic Imaging and Tomography*, vol. 10580, p. 105800R, International Society for Optics and Photonics, 2018.

[9] X. Yu, H. Wang, W.-c. Feng, H. Gong, and G. Cao, "An enhanced image reconstruction tool for computed tomography on gpus," in *Proceedings of the Computing Frontiers Conference*, (Siena, Italy), pp. 97–106, ACM, 2017.

[10] "Human Brain Project Platforms." `https://www.humanbrainproject.eu/en/hbp-platforms/hbp-joint-platform/`.

[11] "Breaking Down Borders - What Blockchain Can Do For Healthcare." `https://www.medtronic.com/content/dam/medtronic-com/global/Corporate/Documents/what-blockchain-can-do-for-healthcare_paper_mdt_av_corpmark2.pdf`.

[12] "Medtronic - Blockchain in healthcare." `https://www.medtronic.com/us-en/transforming-healthcare/aligning-value/perspective/blockchain-in-healthcare.html`.

[13] "Can Big Data Analytics Save Billions in Healthcare Costs?." `https://www.forbes.com/sites/sap/2016/02/22/can-big-data-analytics-save-billions-in-healthcare-costs/\#6d580fca5d10`.

[14] "IBM - Blockchain in Healthcare." `https://www.ibm.com/blogs/blockchain/category/blockchain-in-healthcare/`.

[15] OpenMP, "The OpenMP API specification for parallel programming." `https://www.openmp.org/`.

[16] B. R. de Supinski, T. R. Scogland, A. Duran, M. Klemm, S. M. Bellido, S. L. Olivier, C. Terboven, and T. G. Mattson, "The Ongoing Evolution of OpenMP," *Proceedings of the IEEE*, no. 99, pp. 1–16, 2018.

[17] J. Bonilla, L. J. Yebra, and S. Dormido, "Exploiting OpenMP in the Initial Section of Modelica Models (Work in Progress)," in *Proceedings of the 4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools; Zurich; Switzerland; September 5; 2011*, no. 056, pp. 107–112, Linköping University Electronic Press, 2011.

[18] "OpenMP Loop Scheduling." `https://software.intel.com/en-us/articles/openmp-loop-scheduling`.

[19] "NVIDIA GPU." `https://developer.nvidia.com/`.

[20] D. B. Kirk, W. W. Hwu, *Programming Massively Parallel Processors. A Hands on Approach*. Morgan Kaufmann, 2010.

[21] "NVIDIA GeForce GTX 680 Whitepaper." `https://www.nvidia.in/content/PDF/product-specifications/GeForce_GTX_680_Whitepaper_FINAL.pdf`.

[22] NVIDIA Corporation, "NVIDIA CUDA C programming guide," 2018.

[23] NVIDIA, "Streams and Concurrency." `https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf`.

[24] NVIDIA Corporation, "GPU Direct." `https://developer.nvidia.com/gpudirect`.

[25] NVIDIA, "Tesla K40 Board Specification." `https://www.nvidia.com/content/PDF/kepler/Tesla-K40-Active-Board-Spec-BD-06949-001_v03.pdf`.

[26] NVIDIA, "GTX1060 Board Specification." `https://www.nvidia.it/geforce/products/10series/geforce-gtx-1060/`.

[27] G. Florimbi, E. Torti, S. Masoli, E. D'Angelo, G. Danese, and F. Leporati, "The human brain project: parallel technologies for biologically accurate simulation of granule cells," *Microprocessors and Microsystems*, vol. 47, pp. 303–313, 2016.

[28] "Human Brain Project - Report." `https://ec.europa.eu/research/participants/portal/doc/call/h2020/fetflag-1-2014/1595110-6pilots-hbp-publicreport_en.pdf`.

[29] P. Dale, "Neuroscience (Third edition)," 2004.

[30] L. Mapelli, S. Solinas, and E. D'Angelo, "Integration and regulation of glomerular inhibition in the cerebellar granular layer circuit," *Frontiers in cellular neuroscience*, vol. 8, p. 55, 2014.

[31] E. M. Izhikevich, "Which model to use for cortical spiking neurons?," *IEEE transactions on neural networks*, vol. 15, no. 5, pp. 1063–1070, 2004.

[32] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *The Journal of physiology*, vol. 117, no. 4, pp. 500–544, 1952.

[33] G. Florimbi, E. Torti, G. Danese, and F. Leporati, "High performant simulations of cerebellar Golgi cells activity," in *25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), 2017*, pp. 527–534, IEEE, 2017.

[34] E. D'Angelo, T. Nieus, A. Maffei, S. Armano, P. Rossi, V. Taglietti, A. Fontana, and G. Naldi, "Theta-frequency bursting and resonance in cerebellar granule cells: experimental evidence and modeling of a slow K+-dependent mechanism," *Journal of Neuroscience*, vol. 21, no. 3, pp. 759–770, 2001.

[35] S. Solinas, T. Nieus, and E. D'Angelo, "A realistic large-scale model of the cerebellum granular layer predicts circuit spatio-temporal filtering properties," *Frontiers in cellular neuroscience*, vol. 4, p. 12, 2010.

[36] "Neuron." https://www.neuron.yale.edu/neuron/.

[37] S. Solinas, L. Forti, E. Cesana, J. Mapelli, E. De Schutter, and E. D'Angelo, "Computational reconstruction of pacemaking and intrinsic electroresponsiveness in cerebellar Golgi cells," *Frontiers in cellular neuroscience*, vol. 1, p. 2, 2007.

[38] T. Nieus, E. Sola, J. Mapelli, E. Saftenku, P. Rossi, and E. D'Angelo, "LTP regulates burst initiation and frequency at mossy fiber–granule cell synapses of rat cerebellum: experimental observations and theoretical predictions," *Journal of neurophysiology*, vol. 95, no. 2, pp. 686–699, 2006.

[39] S. Herculano-Houzel, "The human brain in numbers: a linearly scaled-up primate brain," *Frontiers in human neuroscience*, vol. 3, p. 31, 2009.

[40] "Stratix V Altera FPGA." https://www.intel.com/content/www/us/en/products/programmable/fpga/stratix-v.html.

[41] "Galileo (CINECA)." http://www.hpc.cineca.it/hardware/galileo.

[42] "NVIDIA Tesla K80." https://www.nvidia.com/en-us/data-center/tesla-k80/.

[43] E. D'Angelo, S. Solinas, J. Mapelli, D. Gandolfi, L. Mapelli, and F. Prestori, "The cerebellar Golgi cell and spatiotemporal organization of granular layer activity," *Frontiers in neural circuits*, vol. 7, p. 93, 2013.

[44] E. D'Angelo, A. Antonietti, S. Casali, C. Casellato, J. A. Garrido, N. R. Luque, L. Mapelli, S. Masoli, A. Pedrocchi, F. Prestori, *et al.*, "Modeling the cerebellar microcircuit: New strategies for a long-standing issue," *Frontiers in cellular neuroscience*, vol. 10, p. 176, 2016.

[45] F. Naveros, N. R. Luque, J. A. Garrido, R. R. Carrillo, M. Anguita, and E. Ros, "A Spiking Neural Simulator Integrating Event-Driven and Time-Driven Computation Schemes Using Parallel CPU-GPU Co-Processing: A Case Study.," *IEEE Trans. Neural Netw. Learning Syst.*, vol. 26, no. 7, pp. 1567–1574, 2015.

[46] T. Yamazaki, J. Igarashi, J. Makino, and T. Ebisuzaki, "Real-time simulation of a cat-scale artificial cerebellum on PEZY-SC processors," *The International Journal of High Performance Computing Applications*, p. 1094342017710705, 2017.

[47] P. Gleeson, V. Steuber, and R. A. Silver, "neuroConstruct: a tool for modeling networks of neurons in 3D space," *Neuron*, vol. 54, no. 2, pp. 219–235, 2007.

[48] H. Fabelo, S. Ortega, D. Ravi, B. R. Kiran, C. Sosa, D. Bulters, G. M. Callicó, H. Bulstrode, A. Szolna, J. F. Piñeiro, *et al.*, "Spatio-spectral classification of hyperspectral images for brain cancer detection during surgical operations," *PloS one*, vol. 13, no. 3, p. e0193721, 2018.

[49] N. Sanai and M. S. Berger, "Glioma extent of resection and its impact on patient outcome," *Neurosurgery*, vol. 62, no. 4, pp. 753–766, 2008.

[50] W. Stummer, J.-C. Tonn, H. M. Mehdorn, U. Nestler, K. Franz, C. Goetz, A. Bink, and U. Pichlmeier, "Counterbalancing risks and gains from extended resections in malignant glioma surgery: a supplemental analysis from the randomized 5-aminolevulinic acid glioma resection study," *Journal of neurosurgery*, vol. 114, no. 3, pp. 613–623, 2011.

[51] M. Reinges, H.-H. Nguyen, T. Krings, B.-O. Hütter, V. Rohde, and J. Gilsbach, "Course of brain shift during microsurgical resection of supratentorial cerebral lesions: limits of conventional neuronavigation," *Acta neurochirurgica*, vol. 146, no. 4, pp. 369–377, 2004.

[52] K. Ganser, H. Dickhaus, A. Staubert, M. Bonsanto, C. Wirtz, V. Tronnier, and S. Kunze, "Quantification of brain shift effects in MRI

images," *Biomedizinische Technik. Biomedical engineering*, vol. 42, pp. 247–248, 1997.

[53] W. Stummer, U. Pichlmeier, T. Meinel, O. D. Wiestler, F. Zanella, H.-J. Reulen, A.-G. S. Group, *et al.*, "Fluorescence-guided surgery with 5-aminolevulinic acid for resection of malignant glioma: a randomised controlled multicentre phase III trial," *The lancet oncology*, vol. 7, no. 5, pp. 392–401, 2006.

[54] G. Lu and B. Fei, "Medical hyperspectral imaging: a review," *Journal of biomedical optics*, vol. 19, no. 1, p. 010901, 2014.

[55] S. V. Panasyuk, S. Yang, D. V. Faller, D. Ngo, R. A. Lew, J. E. Freeman, and A. E. Rogers, "Medical hyperspectral imaging to facilitate residual tumor identification during surgery," *Cancer biology & therapy*, vol. 6, no. 3, pp. 439–446, 2007.

[56] L. A. Zherdeva, I. A. Bratchenko, M. V. Alonova, O. O. Myakinin, D. N. Artemyev, A. A. Moryatov, S. V. Kozlov, and V. P. Zakharov, "Hyperspectral imaging of skin and lung cancers," in *Biophotonics: Photonic Solutions for Better Health Care V*, vol. 9887, p. 98870S, International Society for Optics and Photonics, 2016.

[57] H. Akbari, K. Uto, Y. Kosugi, K. Kojima, and N. Tanaka, "Cancer detection using infrared hyperspectral imaging," *Cancer science*, vol. 102, no. 4, pp. 852–857, 2011.

[58] H. Akbari, L. Halig, D. M. Schuster, B. Fei, A. Osunkoya, V. Master, P. Nieh, and G. Chen, "Hyperspectral imaging and quantitative analysis for prostate cancer detection," *Journal of biomedical optics*, vol. 17, no. 7, p. 076005, 2012.

[59] Z. Liu, H. Wang, and Q. Li, "Tongue tumor detection in medical hyperspectral images," *Sensors*, vol. 12, no. 1, pp. 162–174, 2011.

[60] D. N. Louis, A. Perry, G. Reifenberger, A. Von Deimling, D. Figarella-Branger, W. K. Cavenee, H. Ohgaki, O. D. Wiestler, P. Kleihues, and

D. W. Ellison, "The 2016 World Health Organization classification of tumors of the central nervous system: a summary," *Acta neuropathologica*, vol. 131, no. 6, pp. 803–820, 2016.

[61] E. Torti, A. Fontanella, G. Florimbi, F. Leporati, H. Fabelo, S. Ortega, and G. Callico, "Acceleration of brain cancer detection algorithms during surgery procedures using GPUs," *Microprocessors and Microsystems*, 2018.

[62] H. Fabelo, S. Ortega, R. Guerra, G. Callicó, A. Szolna, J. F. Piñeiro, M. Tejedor, S. López, and R. Sarmiento, "A Novel Use of Hyperspectral Images for Human Brain Cancer Detection Using in-Vivo Samples," in *Proceedings of the International Joint Conference on Biomedical Engineering Systems and Technologies*, BIOSTEC 2016, (Rome, Italy), pp. 311–320, 2016.

[63] G. Florimbi, H. Fabelo, E. Torti, R. Lazcano, D. Madroñal, S. Ortega, R. Salvador, F. Leporati, G. Danese, A. Báez-Quevedo, G. M. Callicó, E. Juárez, C. Sanz, and R. Sarmiento, "Accelerating the K-Nearest Neighbors Filtering Algorithm to Optimize the Real-Time Classification of Human Brain Tumor in Hyperspectral Images," *Sensors*, vol. 18, no. 7, p. 2314, 2018.

[64] H. Fabelo, S. Ortega, R. Lazcano, D. Madroñal, G. M Callicó, E. Juárez, R. Salvador, D. Bulters, H. Bulstrode, A. Szolna, *et al.*, "An intraoperative visualization system using hyperspectral imaging to aid in brain tumor delineation," *Sensors*, vol. 18, no. 2, p. 430, 2018.

[65] "OpenCV Library." `https://opencv.org/`.

[66] NVIDIA, "cuBLAS Library Documentation." `http://docs.nvidia.com/cuda/cublas/index.html`.

[67] R. Lazcano, D. Madroñal, R. Salvador, K. Desnos, M. Pelcat, R. Guerra, H. Fabelo, S. Ortega, S. Lopez, G. Callico, *et al.*, "Porting

a PCA-based hyperspectral image dimensionality reduction algorithm for brain cancer detection on a manycore architecture," *Journal of Systems Architecture*, vol. 77, pp. 101–111, 2017.

[68] NVIDIA, "cuBLAS Library Documentation." `https://docs.nvidia.com/cuda/cusolver/index.html`.

[69] D. Madroñal, R. Lazcano, R. Salvador, H. Fabelo, S. Ortega, G. Callico, E. Juarez, and C. Sanz, "SVM-based real-time hyperspectral image classifier on a manycore architecture," *Journal of Systems Architecture*, vol. 80, pp. 30–40, 2017.

[70] K. Huang, S. Li, X. Kang, and L. Fang, "Spectral–spatial hyperspectral image classification based on KNN," *Sensing and Imaging*, vol. 17, no. 1, p. 1, 2016.

[71] Y. Tarabalka, J. A. Benediktsson, and J. Chanussot, "Spectral–spatial classification of hyperspectral imagery based on partitional clustering techniques," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 47, no. 8, pp. 2973–2987, 2009.

List of publications

Articles in peer reviewed journals

- Florimbi G., Torti E., Masoli S., D'Angelo E., Danese G. and Leporati F., "The Human Brain Project: Parallel technologies for biologically accurate simulation of Granule cells", *Microprocessors and Microsystems*, vol. 47, part B, pp. 303-313, November 2016.

- Torti E., Fontanella A., Florimbi G., Leporati F., Fabelo H., Ortega S. and Callicó G.M., "Acceleration of brain cancer detection algorithms during surgery procedures using GPUs", *Microprocessors and Microsystems*, vol. 61, pp. 171-178, September 2018.

- Florimbi G., Fabelo H., Torti E., Lazcano R., Madroñal D., Ortega S., Salvador R., Leporati F., Danese G., Báez-Quevedo A., Callicó G.M., Juárez E., Sanz C. and Sarmiento R., "Accelerating the K-Nearest Neighbors Filtering Algorithm to Optimize the Real-Time Classification of Human Brain Tumor in Hyperspectral Images", *Sensors*, vol. 18(7), July 2018.

Contributions to conference proceedings

- Florimbi G., Torti E., Danese G. and Leporati F., "High performant simulations of cerebellar Golgi cells activity", *25th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, St. Petersburg (Russia), March 6-8, 2017.